Kriging Convolutional Networks

Gabriel Appleby,*  Linfeng Liu,*  Li-Ping Liu
Department of Computer Science, Tufts University
{Gabriel.Appleby, Linfeng.Liu, Liping.Liu}@tufts.edu

Abstract

Spatial interpolation is a class of estimation problems where locations with known values are used to estimate values at other locations, with an emphasis on harnessing spatial locality and trends. Traditional kriging methods have strong Gaussian assumptions, and as a result, often fail to capture complexities within the data. Inspired by the recent progress of graph neural networks, we introduce Kriging Convolutional Networks (KCN), a method of combining advantages of Graph Neural Networks (GNN) and kriging. Compared to standard GNNs, KCNs make direct use of neighboring observations when generating predictions. KCNs also contain the kriging method as a specific configuration. Empirically, we show that this model outperforms GNNs and kriging in several applications.

Spatial data is ubiquitous in a wide variety of fields such as ecology (Fink et al. 2010), economics (Gao and Liu 2014), and meteorology (Xingjian et al. 2015). A common task within these fields is to estimate values at target locations from nearby known values. Improving these estimations should provide clear benefits for these applications. Estimation techniques tailored to spatial data must leverage the fact that every data point is associated with a location. Most importantly, these techniques should be able to capture the spatial correlation among these locations.

In many fields, the most prevalent method for spatial data modeling is kriging (Cressie 1991). The fundamental assumption of kriging is that observations at locations are from an underlying Gaussian process. After estimating the variogram, which is essentially the strength of spatial correlations between data points, kriging uses a linear interpolation of observed values to predict the value at a new location. The kriging prediction is the best linear unbiased estimator for spatial points given its Gaussian assumption. However, this assumption is quite constritive, as data in many applications are not from a Gaussian distribution. For example, we will show in our experiments that this assumption leads to poor performance when estimating integer counts that contain a significant fraction of zeros.

Researchers also use flexible machine learning algorithms (Hengl et al. 2018) for spatial data modeling. Given the huge success of GNNs and the similarity between spatial data and graph data, researchers have started to apply Graph Neural Networks (GNN) (Wu et al. 2019) to spatial data (Li et al. 2017; Yu, Yin, and Zhu 2017; Zhu and Liu 2018; Yan et al. 2019). GNNs were first developed for explicit graph data, but can model any data that can be transformed into a graph either by their spatial vicinity or their physical connections (e.g. routes). The main idea is to propagate information along graph edges, so graph nodes can share information during the learning process. GNNs are relatively generic, and can find nonlinear relationships between the inputs, hidden layers, and neighborhood information of each node. By design, GNNs are more flexible than kriging.

However, kriging has an advantage over GNNs: kriging directly uses observed training labels to predict the label of a new data point. In comparison, there is no straightforward way to feed training labels as input to a GNN. It is not feasible to directly feed training labels as part of the input because the GNN will directly output the given label of a training data point and learn nothing. Furthermore, spatial data modeling requires inductive learning – the model needs to be able to make predictions for new locations that are not in the graph formed by training data. While kriging is intrinsically inductive, only a few GNNs such as GraphSAGE (Hamilton, Ying, and Leskovec 2017) can work inductively.

Inspired by these two observations, we develop a new model, the Kriging Convolutional Network (KCN), as an improvement to GNNs. The KCN is still a type of GNN. However, it does not form a single large graph over all data points. Every time a KCN fits the label of a data point (call it the center), it forms a small graph over the center and its neighboring training data points. These neighbors are the K nearest neighbors according to a distance metric. In the input to the KCN, we hide the label of the center node. The input consists of feature vectors for all nodes in the graph (K + 1 nodes), as well as the labels of the neighbors. The KCN also needs the adjacency matrix of the graph, which is defined to be the spatial kernel matrix or a normalized version of that. The target value of the KCN is the label of the center node.

*Equal contribution. The first two authors are arranged by the alphabet order

Copyright © 2020, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.
We iterate over all of the training data, treating each node as the center to train the KCN model. The KCN uses the same structure to predict the label of a new data point.

The KCN combines the best parts of both models. In comparison to the GNN, it is able to directly leverage training labels in prediction, and no re-training is necessary when new data points are introduced. In contrast to kriging, the KCN is more versatile. On a large dataset where overfitting is not an issue, the KCN has a clear advantage over kriging. Even though the KCN’s underlying mechanisms are very different from kriging, our theoretical analysis reveals a deep connection between the two models. In fact, with a special configuration, the KCN can emulate kriging.

In summary, this work has three contributions:

- the development of the KCN, which is a GNN that directly uses training labels for prediction;
- the theoretical result showing that the KCN approximately recovers local universal kriging; and
- empirical studies indicating the KCN’s advantage over baseline models.

Related Work

Kriging (Cressie 1991) has been widely used in spatial data modeling. Using kriging to model non-Gaussian data is often accomplished through careful transformation of labels (Saito and Goovaerts 2000). However, it is not always feasible to transform a variable to be Gaussian (Dance 2018). There are many variants of kriging, of which universal kriging is the most appropriate for the setting above. Universal kriging has the following model assumption (Eq. 3.4.2 in (Cressie 1991)).

\[ y_i = \beta^T x_i + \epsilon(s_i), \quad i = 1, \ldots, n, * \]  

Here \( \beta \) is the coefficient vector, \( \epsilon(\cdot) \) is a zero-mean random process with variogram \( 2\gamma(\cdot) \). The variogram \( 2\gamma(\cdot) \), which specifies the spatial correlation between data points, is a function of spatial distance: \( 2\gamma(\|s_i - s_j\|) = \mathbb{E}[(\epsilon(s_i) - \epsilon(s_j))^2] \). The variogram often takes a special form with its parameters estimated from the data. With this model assumption, kriging minimizes the expected squared error, \( E_{y^*}[(\hat{y} - y^*)^2] \), in closed form. Then the prediction \( y^* \) of universal kriging is \( \hat{y} = \lambda^T y \) with

\[ \lambda = \Gamma^{-1}(\beta - B X^T \Gamma^{-1} y), \]  

with \( B = X(X^T \Gamma^{-1} X)^{-1} \), \( \Gamma = [\gamma(\|s_i - s_j\|)]_{i,j=1}^n \), and \( \gamma = [\gamma(\|s_i - s_j\|)]_{i=1}^n \).

Note that kriging uses known training labels as well as all features as the input to make the prediction. Despite its complex form, kriging has a subtle relation with the KCN model proposed later.

Graph Convolutional Networks

Suppose we have a graph \( G = (V, E) \), where \( V = \{1, \ldots, M\} \) is set of data points, and \( E \) is the edge set. Each data point \( i \in V \) has a feature vector \( x_i \) and a label \( y_i \). All feature vectors and labels are collectively denoted by \( \tilde{X} \) and \( \hat{y} \). Let \( A \) denote the adjacency matrix of the graph, and \( \tilde{A} \) denote the normalized adjacency matrix,

\[ \tilde{A} = D^{-\frac{1}{2}}(A + I)D^{-\frac{1}{2}}, \]  

with \( D = \text{diag}(A_1 + 1) \) being the degree matrix plus one. Then a GCN (Kipf and Welling 2016) takes \( \tilde{A} \) and \( \tilde{X} \) as the input and fits known labels in \( \hat{y} \) as the target. The GCN consists of \( L \) GCN layers. Each GCN layer \( \ell \) takes an input \( H_{\ell-1} \in \mathbb{R}^{n \times d_{\ell-1}} \) and outputs a matrix \( H_{\ell} \in \mathbb{R}^{n \times d_\ell} \). The layer is parameterized by a matrix \( W^{\ell} \) with size \( d_{\ell-1} \times d_\ell \). Formally, the GCN is defined by

\[ H^0 = \tilde{X}, \]  
\[ H^{\ell} = \sigma(\tilde{A}H^{\ell-1}W^{\ell}), \quad \ell = 1, \ldots, L \]  
\[ \hat{y} = H^L. \]  

Here \( \sigma(\cdot) \) is a non-linear activation function.

Usually only part of the node labels in \( \hat{y} \) are observed, and the task is to predict unknown labels of other nodes. A GCN defines its training loss based on known labels and predicts unknown labels as corresponding entries in \( \hat{y} \). To
apply a GCN to the previous task, we form a graph over data points based on their locations \((s_i, s_d)\), e.g., nearest-neighbor graph. Then each feature vector and its label are attached to the corresponding graph node. The prediction for the new location \( \alpha^* \) is just \( \hat{y}_s \) taken from \( y_s \).

**Kriging Convolution Network**

In this section, we develop a new learning model that directly uses training labels as the inputs for predictions. We call this model a Kriging Convolution Network (KCN).

We will first demonstrate how a KCN will be used for prediction. Let's treat a KCN model as a function \( KCN(\cdot; \theta) \) parameterized by \( \theta \). When predicting the label of a new data point \((s, x_s)\), the model ideally should use all information available to make the prediction, that is, \( \hat{y}_s = KCN(s, X, y, s, s, x_s) \). However, it is not feasible to consider all of the training points for just one prediction. It is not necessary either, because data points far from \( s \), often have little influence over \( y_s \) in many spatial problems. Therefore, we use the \( K \) nearest neighbors of the new data point as the input. Denote the index set of these neighbors as \( \alpha_s \subset \{1, \ldots, N\} \), then the predictive function becomes

\[
\hat{y}_s = KCN(s_{\alpha_s}, X_{\alpha_s}, y_{\alpha_s}, s, s, x_s). \tag{7}
\]

To train the model, we treat every training point \( i \) as a test point and fit its training label \( y_i \). The model’s output, \( \hat{y}_i \), is compared against the true label. The difference of the two is measured by some loss function \( \text{loss}(y_i, \hat{y}_i) \). The learning objective of the model is to minimize the summation of all training losses

\[
\min_{\theta} \sum_{i=1}^{N} \text{loss}(y_i, \hat{y}_i), \quad \hat{y}_i = KCN(s_{\alpha_i}, X_{\alpha_i}, y_{\alpha_i}, s_i, x_i). \tag{8}
\]

Here \( \alpha_i \) is the set of neighbors of \( i \) in the training set.

Now we construct the network architecture of the KCN, i.e., the function \( KCN(s_{\alpha_i}, X_{\alpha_i}, y_{\alpha_i}, s_i, x_i) \). Instead of using locations, \( s_i \) and \( s_{\alpha_i} \), as features, we define a complete graph over the data points \( i \) and its neighbors and then use a GCN to construct the predictive model. Denote \( \beta_i = \{i\} \cup \alpha_i \) as the set containing the data point \( i \) and its neighbors. We first define a graph over \( \beta_i \) by constructing its adjacency matrix \( A \) from a Gaussian kernel,

\[
A_{jk} = \exp \left( -\frac{1}{2\phi^2} \| s_j - s_k \|_2^2 \right), \quad \forall j, k \in \beta_i. \tag{9}
\]

Here \( \phi \) is the kernel length, which is a hyperparameter. In this graph, the edge \((j, k)\) has a large weight when \( j \) and \( k \) are near each other and vice versa.

Next we define the feature input to the GCN. The input should include features, \( x_i \) and \( X_{\alpha_i} \), and neighboring labels \( y_{\alpha_i} \). Incorporating this information into a matrix will require a bit of care. We place \( y_{\alpha_i} \) and a zero in place of \( y_i \) into a vector with length \((K + 1)\), so the model has no access to \( y_i \). We also use an indicator vector \( e \) to indicate that the instance \( i \) is the one to be predicted. Then the GCN input is expressed by a matrix \( H^0 \) with size \((K + 1) \times (2 + d)\).

\[
H^0 = \begin{bmatrix} 0 & 1 & x_i^\top \\ y_{\alpha_i} & 0 & X_{\alpha_i} \end{bmatrix}. \tag{10}
\]

The locations \( s_{\beta_i} \) can be included in the feature matrix \( X \) as features if there is reason to suspect spatial trends.

Then the KCN model is defined to be a GCN followed by a dense layer. The KCN is formally defined as

\[
H^L = GCN(A, H^0, W^1, \ldots, W^L); \quad \hat{y}_i = \sigma(e^\top H^L w_{\text{den}}); \quad \text{loss}(y_i, \hat{y}_i).
\]

The locations \( s_{\beta_i} \) are the adjacency matrix and the input feature matrix constructed from the neighborhood of \( i \). Note that every data point \( i \) gets its own \( A \) and \( H^0 \), whose index \( i \) is omitted for notational simplicity. The vector \( e \) is the indicator vector for \( i \); it takes the first vector of \( H^L \), corresponding to \( i \), as the input to the dense layer. The dense layer allows for a final transformation of the data without interference from neighbors.

The KCN parameters are all weight matrices, \( \theta = \{W^1, \ldots, W^L, w_{\text{den}}\} \). We train the KCN model by minimizing the loss in (8). Then we can predict the label of a new data point using its features and neighbors in the training set. Algorithm 1 summarizes the training procedure of the KCN.

Compared to local kriging, which only uses nearest neighbors for kriging, the KCN uses the same input. However, the KCN is much more flexible. When the training set is large enough such that the overfitting issue is less of a concern, the KCN model has clear advantages.

Compared to the direct application of a GCN on spatial data, a KCN is able to use labels from neighbors directly. Furthermore, a KCN does not need to use the test data points to form the graph. Therefore, it does not need to re-train the model when there is a new batch of test data points.

The KCN is also similar to the KNN classifier but is much more powerful: while the KNN simply averages the labels of neighbors, the KCN uses a neural network as the predictive function.
KCN with Graph Attention

The recent success of attention mechanism on GNNs inspires us to try the Graph Attention network (GAT) (Veličković et al. 2018) as the predicting model. The original GAT model computes attention weights with a neural network; it also requires that the attention weights of a node’s neighbors sum up to 1. Here we use the dot-product self-attention (Vaswani et al. 2017) so that the model has a choice to fall back on the GCN model.

Suppose the input feature at the $\ell$-th layer of the GCN is $H^{\ell-1}$, then we compute an attention matrix $U$ by

$$P = H^{\ell-1}W_{att}, \quad M = \sigma(PP^\top), \quad \Lambda = \text{diag}(M), \quad U = \Lambda^{-\frac{1}{2}}MA^{-\frac{1}{2}}. \quad (13)$$

Here $W_{att}$ is the weight matrix for the attention mechanism. It projects input features into a new space. Then the attention weights are decided by inner products between features in this new space. We normalize the attention matrix so that the diagonal elements of $U$ are always one.

In each layer $\ell$, we get an attention matrix $U_{\ell}$ as above. Then we use $A^{\text{att}} = A \odot U_{\ell}$ as the new adjacency matrix used in layer $\ell$. The actual computation is

$$H^{\ell} = \sigma \left( A^{\text{att}} H^{\ell-1} W \right), \quad \ell = 1, \ldots, L \quad (14)$$

We call this new model the KCN-att. When the matrix $W_{att}$ has small weights, then $U$ approaches a matrix with all entries being one. In this case, the KCN-att becomes similar to the KCN. When the matrix $W_{att}$ has large weights, then $U$ tends to approach the identity matrix, and then the KCN-att tends to reduce neighbors’ influence.

KCN based GraphSAGE

We also use GraphSAGE (Hamilton, Ying, and Leskovec 2017) as the predictive model of the KCN given that GraphSAGE performs well on several node classification tasks. GraphSAGE cannot use a weighted graph, so we treat the graph over the neighborhood of $i$ as a complete graph. Let $H^{\ell-1} = \{h_k^{\ell-1} : k \in \beta_i\}$ be the input to the GraphSAGE layer $\ell$, then the layer computes its output $H^{\ell}$ as follows.

$$g_j^{\ell} = \text{AGG}(\{h_k^{\ell-1} : k \in \beta_i, k \neq j\}), \quad \forall j \in \beta_i \quad (15)$$

$$h_j^{\ell} = \sigma(W_1g_j^{\ell-1} + W_2h_j^{\ell-1}), \quad \forall j \in \beta_i \quad (16)$$

$$H^{\ell} = \{h_j^{\ell} : h_j^{\ell} \| h_j^{\ell} \|_2 : \forall j \in \beta_i\} \quad (17)$$

The function $\text{AGG}(\cdot)$ aggregates a list of vectors into one. We use the max-pooling aggregator, one the three aggregators proposed in the original work (Hamilton, Ying, and Leskovec 2017).

$$\text{AGG}(H^{\ell-1}_{\beta_i \cup j}) = \max(\sigma(W_{\text{pool}}h_k^{\ell-1} + b), k \in \beta, k \neq j)$$

Here $\max$ takes the element-wise max values over a list of vectors. We refer to this model as the KCN-sage.

Analysis

Computation Complexity

The time complexity of the KCN and the two variants includes nearest-neighbor search and network training. In order to find the $K$ nearest neighbors we utilize a KD tree, which takes $O(N \log(N))$ time to build. Here we treat the dimensionality of spatial coordinates as a constant because it is usually a small number (2 or 3). Querying a single data point in the tree takes time $O(K \log(N))$, and searching neighbors for all data points takes a total of $O(NK \log(N))$ time.

When we train the model on a single instance, the computation of the adjacency matrix takes time $O(K^2)$. The computation within each layer takes time $O(K^2 d_{max})$, with $d_{max}$ being the largest dimensionality of hidden layers. The forward computation and backpropagation for one instance takes time $O(K^2 L d_{max})$, and one training epoch takes time $O(NK^2 L d_{max})$.

Relation to Kriging

The KCN is a flexible model and approximately includes local kriging as a special case. This fact is shown by the following theorem.

**Theorem 1:** Assume the variogram of a kriging model satisfies $2\gamma(0) > 0$. Also assume $X = [x_s, X_{\alpha}, \tau]^\top$ has full column rank. Then there exists a set of special parameters and activations with which a KCN makes the same prediction as the kriging prediction, i.e. $\hat{y}^{\text{KCN}} = \hat{y}^{\text{kriging}}$.

**Proof sketch:** Let $\Gamma$ be the covariance matrix corresponding to the new data point and training data point.

$$\hat{\Gamma} = \begin{bmatrix} 0 & \gamma & \Gamma \end{bmatrix} \quad (18)$$

Here $\gamma$ and $\Gamma$ are semivariograms defined in the same way as kriging. To approximate kriging, we set the KCN to have one convolutional layer and a dense layer. We set

$$\tilde{A} = \hat{\Gamma}^{-1} - \hat{\Gamma}^{-1}X(X^\top \hat{\Gamma}^{-1}X)^{-1}X^\top \hat{\Gamma}^{-1} \quad (19)$$

as the “normalized adjacency matrix” and directly use it to multiply the hidden input. We consider a 1-layer GCN with a special activation function $\sigma_{\text{div}}(\cdot)$. The first row of the GCN output is $e^\top \tilde{A}H^L = \sigma_{\text{div}}(e\tilde{A}H^LW)$. In the Appendix we show $e\tilde{A} = [z^{-1}, -z^{-1}\lambda^\top]^\top$, then

$$e\tilde{A}e^\text{H}^0 = \left[\begin{array}{c} -z^{-1}\lambda^\top y_{\alpha}, \quad z^{-1}, \quad z^{-1}(x_s - \lambda^\top X_{\alpha})\end{array}\right].$$

Here $z$ is a scalar, and $\lambda$ is the kriging coefficient defined in (2). Take the first two elements of $e\tilde{A}H^0$ and denote it as $u = [-z^{-1}\lambda^\top y_{\alpha}, z^{-1}]$. We can see that $-u_1/u_2 = \lambda^\top y_{\alpha} = \gamma_k^{\text{kriging}}$. If the rest of the network is able to emulate $f(u) = -u_1/u_2$, then $\gamma_{\text{KCN}}^{\text{kriging}}$ is the same as $\gamma_k^{\text{kriging}}$.

Now consider the network computation on $u$. It is a combination of the first two rows of $W^1$, the GCN activation, and then the dense layer, so it can be viewed as a two-layer feedforward network applied to $u$. If special activations are allowed, such as step functions and the logarithm, then the two-layer neural network can realize the function $-u_1/u_2$. With normal network settings, the network can also approximate the function due to the universal approximation theorem. \(\square\)

\(^3\)The value $2\gamma(0)$ is called the nugget of the variogram, which is usually greater than zero.
Experiment

We evaluate our methods on three tasks: bird count modeling, restaurant rating regression, and precipitation regression. We use kriging, Random Forest, GCN, and GraphSAGE as baselines.

Experiment setup

Kriging: we use the implementation of kriging within Automap (Hiemstra et al. 2008). Automap essentially automates the process of Kriging by automatically fitting variograms and testing a number of different models. In all of our experiments, Automap tests spherical, exponential, gaussian, matern, and stein variograms and picks the best one based on the smallest residual sum of squares. Since all of the datasets have a large number of data points, we use local kriging and only consider the closest 100 points.

Random Forest: Hengl et al. (2018) use Random Forest to make predictions for spatial data. For each data point, the algorithm calculates the distances between that point and all training points. These distances are then used as the feature vector of that data point. This algorithm does not scale to very large datasets, so we downsample the training set to a size of 1000. We use the implementation of Random Forest (Wright and Ziegler 2017), and method of tuning (Probst, Wright, and Boulesteix 2018) used by the authors of Hengl et al. (2018). The implementation tunes four hyperparameters of Random Forest: the number of trees to use, the number of variables to consider at a node split, the minimal node size, and the sample fraction when training each tree.

GCN: we modify Kipf’s implementation of (Kipf and Welling 2016) for regression problems. Before we run the GCN on spatial data, we first build an undirected graph over data points: we connect two data points if one is among the other’s K nearest neighbors. We only consider a GCN with two hidden layers. We tune the hyper-parameters of the GCN in the same way as we tune the KCN and the KCN-att below.

GraphSAGE: we implement GraphSAGE with the Spektral graph deep learning library. For each experiment, we build an undirected graph in the same way as GCN. Then we train a two hidden layer GraphSAGE whose hyperparameters are tuned as below.

KCN & KCN-att & KCN-sage: the three models use two hidden layers respectively. We tune the following hyperparameters: hidden sizes $\in \{(20, 10), (10, 5), (5, 3)\}$, dropout rate $\in \{0, 0.25, 0.5\}$, and kernel length $\in \{1, 5, 10, 50\}$.

Note that GraphSAGE and KCN-sage do not consider edge weights of the adjacency matrix, so we do not tune kernel length for them. We also employed early stopping to decide the number epochs.

Bird count modeling

One application of KCNs is modeling bird count data from the eBird project (Fink et al. 2010), which contains over one billion records of bird observation events. Modeling bird data from the eBird project provides an opportunity to deepen our understanding of birds as part of the ecosystem. In this experiment, we model the distribution of wood thrush in June, which is of great interest to ornithologists (Johnston et al. 2019). Figure 1 shows a picture of a wood thrush and the distribution of observed counts over the eastern US.

We restrict our data to a subset of records of wood thrush in June 2014. Each record has a GPS location, a count of wood thrushes observed, and a list of features such as observation time, count type (stationary count, traveling count, etc.), effort hours, and effort area. After removing 583 records with uncertain counts or counts over 10, we get 107,246 records to form our dataset. Bird counts in this dataset are highly sparse: only 11,468 records (fraction of 0.11) have positive counts. We split the dataset into a training set and a test set by 1:1.

When we test our models and baselines, we consider two evaluation metrics. The first one is mean squared error (MSE), so we have a fair comparison with Kriging, the minimization objective of which is the mean squared error. The second one is negative log-likelihood. We use a zero-inflated Poisson distribution (Lambert 1992) as the predictive distri-


<table>
<thead>
<tr>
<th>methods</th>
<th>Kriging</th>
<th>RF</th>
<th>GCN</th>
<th>GraphSAGE</th>
<th>KCN</th>
<th>KCN-att</th>
<th>KCN-sage</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSE</td>
<td>1.56 ± 0.85</td>
<td>0.68 ± 0.03</td>
<td>0.70 ± 0.02</td>
<td>0.53 ± 0.01</td>
<td>0.50 ± 0.01</td>
<td>0.49 ± 0.01</td>
<td><strong>0.44 ± 0.01</strong></td>
</tr>
<tr>
<td>NLL</td>
<td>n.a.</td>
<td>n.a.</td>
<td>1.82 ± 0.00</td>
<td>1.73 ± 0.00</td>
<td>1.60 ± 0.00</td>
<td>1.58 ± 0.00</td>
<td><strong>1.51 ± 0.00</strong></td>
</tr>
</tbody>
</table>

Table 1: Experiment results on the bird count dataset. Performances are measured by the mean squared error and the negative log likelihood of predictions. Smaller values are better.

\[ p(y) = \begin{cases} 
(1 - \expit(u)) + \text{poisson}(y = 0) & \text{if } y = 0, \\
\text{poisson}(y = 0) & \text{if } y > 0. 
\end{cases} \]

(20)

Table 1 shows the performance of the KCN, KCN-att, KCN-sage, and baseline models. From this table, we can see that the three KCN models significantly outperform baseline methods. We also observed that GraphSAGE based methods have better performance than the GCN based methods, we speculate it is because of the concatenation operation (plays a role similar to a skip link) used in the GraphSAGE. Given that bird counts are highly non-Gaussian, we do not expect kriging to perform very well. Random Forest gets much better performance than kriging, but it overly smooths the training data given the small number of training points it can use. The KCN and KCN-att achieve similar performances.

We also study the performances of the GCN, GraphSAGE, KCN, KCN-att, and KCN-sage when different numbers of neighbors are used to form the graph. Figure 2 shows performance values of the five models using different numbers. The GCN perform poorly when the number of neighbors is small in the construction of the graph. In this case, a test point might only connect to another test data point, then the message propagation between two test points is not helpful. GraphSAGE is robust to the number of neighbors. In KCN models, a data point has \( K \) training points as direct neighbors, so KCN models can make better use of the training data in this sense. When a KCN model uses zero neighbors, it is equivalent to a fully connected neural network. In this case, its performance deteriorates significantly, which indicates that spatial correlation exists in the data. KCN, KCN-att, and KCN-sage only need a small number of neighbors to perform well. We speculate that a bird or its nest can be observed only in a small spatial range, so the correlation between near sites is strong but diminishes quickly as the distance increases. The KCN-att performs slightly better than the KCN because it is able to use observatory features to decide whether a neighboring count is from a similar situation or not.

**Restaurant rating regression**

Yelp is a popular website, which allows users to rate and provide information about businesses. They have hosted an extensive collection of these business ratings and attributes for download. In this experiment, we only consider the restaurants within that dataset. Each restaurant has a GPS location and an average rating rounded to the nearest .5, from 0 to 5. Additionally, we choose 13 related attributes from the dataset, all but one of which is categorical. We turn these categorical covariates into 30 indicator variables. These indicators give information about restaurant attributes such as whether it serves alcohol, and whether it takes credit card. After we drop any rows where the ratings, coordinates, or number of reviews is NA, we obtain 188,586 restaurants. We then split the data 1:1 to form a training and test set.

Table 2 shows the experimental results on this dataset. The KCN, KCN-att, and KCN-sage improve the performance of their corresponding baseline models. This task is hard: the features do not seem to be very useful, and the spatial correlations are weak. It is actually hard to overfit the labels with a standard feedforward neural network. Ratings are correlated in a strange way since it is normal that good and bad restaurants are often mixed in the same area. The bad performance of the kriging method can be explained by the fact that the correlations between ratings are highly non-Gaussian.

**Precipitation regression**

The National Oceanic and Atmospheric Administration keeps detailed records of precipitation levels across the United States. One such dataset provides monthly average precipitation in inches from 1981 to 2010 across the US. We average the precipitation level in May for 8,832 stations. We then take the log of these average precipitation values as target values for the regression task. Essentially, we assume a log-normal distribution of precipitation levels. Finally we have a target value, coordinates of each station, and one feature (the elevation) of each station. Data are split 1:1 as a training and testing. Figure 3 shows the data distribution over the US.

Table 3 summarizes the experimental results using the mean squared error. The target values in the log-scale are more likely to be from a Gaussian distribution than the previous two datasets, so the kriging method performs relatively well compared to other methods. The Random Forest method only uses 1000 data points as training data, so it
omits a lot of detailed variations. The GCN models perform poorly on this dataset. One reason is that there are not many features for the GCN to learn. The GCN model becomes more like a “generative” model that generates correlated labels from hidden values. Compared to GNN baselines, the KCN models benefit in particular from using neighboring labels because the KCN models work more like a discriminative model. Note that discriminative models often outperform generative models in supervised learning tasks.

**Conclusion**

In this work, we have introduced the Kriging Convolutional Network, a novel approach to modeling spatial data. Like kriging, the KCN model directly uses training labels in the prediction. However, by employing GNNs as backbone predictive models, the KCN is far more flexible than kriging. We have tried the GCN, GAT, and GraphSAGE as the predictive model of the KCN. The empirical study shows that KCN has significant performance improvement over kriging, the KCN model directly uses training labels in the prediction. However, by employing GNNs as backbone predictive models, the KCN is far more flexible than kriging. Our analysis also reveals that the KCN has the ability to emulate kriging models. This connection further indicates the flexibility of the proposed KCN model.

**Appendix: Detailed Proof of Theorem 1**

We need to derive $\mathbf{e}\mathbf{A}$, where $\mathbf{e} = [1, 0]^T$, $\mathbf{A}$ is “normalized adjacency matrix” defined in (19), and $\mathbf{X} = [x_s, X^T]^T$ are feature vectors. We create the following shorthand notations to facilitate our derivation.

$$t = -\gamma \Gamma^{-1} c, \quad a = -\Gamma^{-1} c, \quad c = x_s + X^T a,$$

$$T = X^T \Gamma^{-1} X, \quad B = X T^{-1}, \quad r = c^T T^{-1} c$$

We will show that the first row of $\mathbf{A}$ is

$$e^T A = \begin{bmatrix} z^{-1}, & -z^{-1} \lambda^T \end{bmatrix}$$

with $z$ being a scalar.

By checking (19), we first compute the inverse $\hat{\Gamma}^{-1}$ is

$$\hat{\Gamma}^{-1} = \begin{bmatrix} t^{-1} & t^{-1} a^T \\ t^{-1} \mathbf{a}^T & \Gamma^{-1} + t^{-1} a a^T \end{bmatrix}$$

$$= t^{-1} \begin{bmatrix} 1 \\ a \end{bmatrix} [1, a^T] + \begin{bmatrix} 0 & 0^T \\ 0 & \Gamma^{-1} \end{bmatrix}.$$

Denote $\mathbf{v}_1 = e \hat{\Gamma}^{-1} = t^{-1} [1, a^T]$.

We then consider the second term in (19). We have

$$\hat{X}^T \hat{\Gamma}^{-1} = t^{-1} c [1, a^T] + [0, X^T \Gamma^{-1}]$$

$$= [t^{-1} c, t^{-1} c a^T + X^T \Gamma^{-1}].$$

Denote $\mathbf{S} = (\hat{X}^T \hat{\Gamma}^{-1} \hat{X})^{-1}$,

$$
\begin{align*}
\mathbf{S} &= (t^{-1} c c^T + \mathbf{T})^{-1} \\
&= \mathbf{T}^{-1} - \mathbf{T}^{-1} c (t + c^T \mathbf{T}^{-1} c)^{-1} c^T \mathbf{T}^{-1}.
\end{align*}
$$

The first line is from the equation $[1, a^T] \hat{X} = c^T$. Denote $\mathbf{v}_2 = e^T \hat{\Gamma}^{-1} X S \hat{X}^T \hat{\Gamma}^{-1}$. Insert the expansion of $\hat{X}^T \hat{\Gamma}^{-1}$, we have

$$\mathbf{v}_2 = t^{-1} c^T \mathbf{S} \begin{bmatrix} t^{-1} c, & t^{-1} c a^T + X^T \Gamma^{-1} \end{bmatrix}.$$

By $c^T \mathbf{S} = t(t+r)^{-1} c^T$ and $w = c^T \mathbf{S} c = r t(t+r)^{-1}$, we have

$$\mathbf{v}_2 = (t + r)^{-1} c^T - tr^{-1} a^T + c^T T^{-1} X^T \Gamma^{-1}.$$

Since $t^{-1} - (t + r)^{-1} t r^{-1} = (t + r)^{-1}$, we have

$$e^T A = \mathbf{v}_1 - \mathbf{v}_2$$

$$= (t + r)^{-1} [1, (a^T - c^T T^{-1} X^T \Gamma^{-1})].$$

Then we expand $a$ and $c$ to get

$$e^T A = (t + r)^{-1} [1, - (\gamma + B x_s - B X \Gamma^{-1} \gamma)^T \Gamma^{-1}].$$

$$= z^{-1} [1, - \lambda^T].$$

Here $z = t + r$.

Since $\mathbf{A}$ are computed from normal matrix operations, its entries are bounded. Therefore, $z \neq 0$.

**Acknowledgments**

This work has been supported in part by Gordon & Betty Moore Foundation, NSF CISE-1908617, and NSF CRII-1850358. We thank all reviewers of this work for their insightful comments.

**References**


