Lernmodul: Eine Einführung in Kriging

Konzeptionelle Grundlagen des Kriging

Von einfachen Modellen zur intelligenten Interpolation

In der modernen ingenieur- und naturwissenschaftlichen Forschung werden Praktiker häufig mit „Black-Box"-Funktionen konfrontiert. Dies sind Systeme oder Simulationen, deren interne Funktionsweise entweder unbekannt oder so komplex ist, dass sie praktisch undurchschaubar ist. Ein gängiges Beispiel ist eine hochpräzise Simulation der numerischen Strömungsmechanik (CFD), bei der Eingaben wie die Flügelgeometrie oder die Strömungsgeschwindigkeit Ausgaben wie Auftrieb und Luftwiderstand erzeugen. Jede Auswertung dieser Black Box kann außerordentlich teuer sein und Stunden oder sogar Tage an Supercomputerzeit in Anspruch nehmen. Wenn das Ziel darin besteht, den Designraum zu erkunden oder ein optimales Design zu finden, ist die Durchführung von Tausenden dieser Auswertungen oft nicht durchführbar.

Diese Herausforderung führt zur Notwendigkeit von **Surrogatmodellen**, auch bekannt als Metamodelle oder Antwortflächenmodelle "Response-Surface". Ein Surrogatmodell ist eine rechengünstige Annäherung an eine teure Black-Box-Funktion. Es wird konstruiert, indem eine kleine Anzahl sorgfältig ausgewählter Auswertungen der wahren Funktion durchgeführt und dann ein mathematisches Modell an diese beobachteten Datenpunkte angepasst wird. Dieses „Modell eines Modells" kann dann Tausende Male zu vernachlässigbaren Kosten ausgewertet werden, was eine effiziente Optimierung, Sensitivitätsanalyse und Erkundung des Designraums ermöglicht.

Eine Brücke zum Kriging: Verständnis von Radialen Basisfunktionen (RBFs)

Eine leistungsstarke und intuitive Klasse von Surrogatmodellen ist das **Modell der Radialen Basisfunktionen (RBF)**. Die grundlegende Idee hinter einem RBF ist es, eine komplexe Funktion als gewichtete Summe einfacherer, gut verstandener Basisfunktionen darzustellen. Jede Basisfunktion ist an einem der bekannten Datenpunkte zentriert, und ihr Wert hängt nur vom Abstand zu diesem Zentrum ab.

Mathematisch hat ein RBF-Modell die Form:

$$\hat{f}(\vec{x}) = \sum_{i=1}^{n} w_i \psi(||\vec{x} - \vec{c}^{(i)}||)$$

wobei $\hat{f}(\vec{x})$ der vorhergesagte Wert an einem neuen Punkt $\vec{x}$ ist, w_i die Gewichte sind, $\vec{c}^{(i)}$ die Zentren der Basisfunktionen (typischerweise die Standorte der bekannten Datenpunkte, $\vec{x}^{(i)}$) und ψ die radiale Basisfunktion selbst ist, die auf dem euklidischen Abstand $||\vec{x} - \vec{c}^{(i)}||$ operiert.

Gängige Wahlen für ψ umfassen die linearen, kubischen, Gauß'schen oder multiquadratischen Funktionen [@Forr08a]. Indem wir fordern, dass das Modell exakt durch alle bekannten Datenpunkte verläuft (ein Prozess, der als Interpolation bezeichnet wird), können wir ein System linearer Gleichungen aufstellen, um die unbekannten Gewichte w_i zu lösen. Dies wird typischerweise in Matrixform geschrieben als:

$$\Psi \vec{w} = \vec{y}$$

wobei Ψ eine Matrix der Auswertungen der Basisfunktionen ist, $\vec{w}$ der Vektor der Gewichte und $\vec{y}$ der Vektor der beobachteten Antworten ist. Das Lösen nach den Gewichten ist dann eine Frage der Matrixinversion: $\vec{w} = \Psi^{-1}\vec{y}$. Die Schönheit dieses Ansatzes liegt darin, dass er ein potenziell hochgradig nichtlineares Modellierungsproblem in ein unkompliziertes lineares Algebraproblem umwandelt [@Forr08a].

Diese Struktur weist eine bemerkenswerte Ähnlichkeit mit anderen Modellierungsparadigmen auf. Die RBF-Formulierung ist funktional identisch mit einem einschichtigen künstlichen neuronalen Netz, bei dem die Neuronen eine radiale Aktivierungsfunktion verwenden. In dieser Analogie ist die Eingabe für jedes Neuron der Abstand von einem Zentrum, die Aktivierungsfunktion des Neurons ist die Basisfunktion ψ, und die Ausgabe des Netzwerks ist die gewichtete Summe dieser Aktivierungen. Diese Verbindung bietet ein nützliches mentales Modell für diejenigen, die mit maschinellem Lernen vertraut sind, und rahmt RBFs nicht als esoterische statistische Technik, sondern als nahen Verwandten von neuronalen Netzen ein, die beide leistungsstarke universelle Funktionsapproximatoren sind.

Einordnung des Kriging

In dieser Landschaft tritt das **Kriging** als eine besonders anspruchsvolle und flexible Art eines RBF-Modells hervor. Ursprünglich aus dem Bereich der Geostatistik durch die Arbeit von Danie G. Krige und Georges Matheron stammend, wurde es entwickelt, um Erzkonzentrationen im Bergbau vorherzusagen [@Forr08a]. Seine Anwendung auf deterministische Computerexperimente wurde von @Sack89a vorangetrieben und ist seitdem zu einem Eckpfeiler des Ingenieurdesigns und der Optimierung geworden.

Im Bereich des maschinellen Lernens ist Kriging besser bekannt als **Gauß-Prozess-Regression (GPR)**. Obwohl sich die Terminologie unterscheidet, ist das zugrunde liegende mathematische Gerüst dasselbe. Kriging unterscheidet sich von einfacheren RBF-Modellen durch seine einzigartige Basisfunktion und seine statistische Grundlage, die nicht nur eine Vorhersage, sondern auch ein Maß für die Unsicherheit dieser Vorhersage liefert.

Die Kernphilosophie des Kriging: Eine stochastische Prozessperspektive

Um das Kriging wirklich zu verstehen, muss man einen konzeptionellen Sprung wagen, der zunächst kontraintuitiv sein kann. Selbst bei der Modellierung eines perfekt deterministischen

Computercodes – bei dem dieselbe Eingabe immer genau dieselbe Ausgabe erzeugt – behandelt das Kriging die Ausgabe der Funktion als eine einzelne Realisierung eines **stochastischen (oder zufälligen) Prozesses**.

Das bedeutet nicht, dass wir annehmen, die Funktion sei zufällig. Stattdessen drücken wir unsere Unsicherheit über den Wert der Funktion an nicht beobachteten Stellen aus. Bevor wir Daten haben, könnte der Wert der Funktion an jedem Punkt alles sein. Nachdem wir einige Punkte beobachtet haben, ist unsere Unsicherheit reduziert, aber sie existiert immer noch überall sonst. Das stochastische Prozessgerüst bietet eine formale mathematische Sprache, um diese Unsicherheit zu beschreiben.

Das Prinzip der Lokalität und Korrelation

Dieser angenommene stochastische Prozess ist nicht völlig unstrukturiert. Er wird von einer **Korrelationsstruktur** bestimmt, die eine grundlegende Annahme über die Welt verkörpert: das Prinzip der Lokalität. Dieses Prinzip besagt, dass Punkte, die im Eingaberaum nahe beieinander liegen, erwartungsgemäß ähnliche Ausgabewerte haben (d. h. sie sind hoch korreliert), während Punkte, die weit voneinander entfernt sind, erwartungsgemäß unähnliche oder unzusammenhängende Ausgabewerte haben (d. h. sie sind unkorreliert). Diese Annahme gilt für die große Mehrheit der physikalischen Phänomene und glatten mathematischen Funktionen, die keine chaotischen, diskontinuierlichen Sprünge aufweisen. Die Korrelation zwischen zwei beliebigen Punkten wird durch eine **Kovarianzfunktion** oder einen **Kernel** quantifiziert, der das Herzstück des Kriging-Modells ist.

Gauß-Prozess-Prior

Speziell nimmt das Kriging an, dass dieser stochastische Prozess ein **Gauß-Prozess** ist. Ein Gauß-Prozess ist eine Sammlung von Zufallsvariablen, von denen jede endliche Anzahl eine gemeinsame multivariate Normalverteilung (MVN) hat. Dies ist eine starke Annahme, da eine multivariate Normalverteilung vollständig durch nur zwei Komponenten definiert ist: einen **Mittelwertvektor ($\vec{\mu}$)** und eine **Kovarianzmatrix (Σ)** [@Forr08a].

Dies ist als der **Gauß-Prozess-Prior** bekannt. Es ist unsere „vorherige Überzeugung“ über die Natur der Funktion, bevor wir Daten gesehen haben. Wir glauben, dass die Funktionswerte an jedem Satz von Punkten gemeinsam gaußverteilt sein werden, zentriert um einen gewissen Mittelwert, mit einer Kovarianzstruktur, die durch den Abstand zwischen den Punkten diktiert wird. Wenn wir Daten beobachten, verwenden wir Bayes'sche Inferenz, um diese vorherige Überzeugung zu aktualisieren, was zu einem **Gauß-Prozess-Posterior** führt. Dieser Posterior ist ebenfalls ein Gauß-Prozess, aber sein Mittelwert und seine Kovarianz wurden aktualisiert, um mit den beobachteten Daten konsistent zu sein. Der Mittelwert dieses posterioren Prozesses gibt uns die Kriging-Vorhersage, und seine Varianz gibt uns ein Maß für die Unsicherheit über diese Vorhersage. Diese statistische Grundlage ist es, die das

Kriging auszeichnet und es ihm ermöglicht, nicht nur zu interpolieren, sondern auch seine eigene Zuverlässigkeit zu quantifizieren.

Die mathematische Architektur eines Kriging-Modells

Um vom konzeptionellen Ansatz des Kriging zur praktischen Umsetzung zu gelangen, ist es unerlässlich, seine mathematischen Komponenten zu verstehen. Dieser Abschnitt analysiert die Architektur des Modells und verbindet konsequent die abstrakte mathematische Notation aus Referenztexten mit den konkreten Variablen, die im bereitgestellten Python-Code verwendet werden.

Glossar der Kriging-Notation

Tabelle 1 dient als Glossar, um die Notation aus [@Forr08a], dem Kochbuch [@bart23iArXiv] und dem in diesem Dokument in Kapitel bereitgestellten Python-Code abzugleichen.

Tabelle 1: Glossar

Mathematisches Symbol	Konzeptionelle Bedeutung	Python-Variable
n	Anzahl der Trainings-/Stichprobenpunkte	`n` oder `X_train.shape`
k	Anzahl der Eingabedimensionen/Variablen	`X_train.shape`
m	Anzahl der Punkte für die Vorhersage	`m` oder `x_predict.shape`
X	$n \times k$ Matrix der Trainingspunkt-Standorte	`X_train`
y	$n \times 1$ Vektor der beobachteten Antworten	`y_train`
$\vec{x}$	Ein neuer Standort für die Vorhersage	Eine Zeile in `x_predict`
Ψ (Psi)	$n \times n$ Korrelationsmatrix der Trainingsdaten	`Psi`
$\vec{\psi}$ (psi)	$n \times m$ Vorhersage-Trainings-Korrelationsmatrix	`psi`
$\vec{\theta}$ (theta)	$k \times 1$ Vektor der Aktivitäts-/Breiten-Hyperparameter	`theta`
$\vec{p}$	$k \times 1$ Vektor der Glattheits-Hyperparameter	Implizit $p = 2$ im Code
μ (mu)	Der globale Mittelwert des stochastischen Prozesses	`mu_hat`

Mathematisches Symbol	Konzeptionelle Bedeutung	Python-Variable
σ^2 (sigma-quadrat)	Die Varianz des stochastischen Prozesses	Im Code nicht explizit berechnet
λ (lambda)	Der Regressions-/Nugget-Parameter	`eps`
$\hat{y}(\vec{x})$	Die Kriging-Vorhersage am Punkt $\vec{x}$	`f_predict`

Der Korrelationskernel: Quantifizierung von Beziehungen

Der Kern des Kriging-Modells ist seine spezialisierte Basisfunktion, auch als Kernel oder Kovarianzfunktion bekannt. Diese Funktion definiert die Korrelation zwischen zwei beliebigen Punkten im Designraum. Die gebräuchlichste Form, und die in unseren Referenztexten verwendete, ist der Gauß'sche Kernel.

Die Kriging-Basisfunktion ist definiert als:

$$\psi(\vec{x}^{(i)}, \vec{x}) = \exp\left(-\sum_{j=1}^{k} \theta_j |x_j^{(i)} - x_j|^{p_j}\right)$$

Diese Gleichung berechnet die Korrelation zwischen einem bekannten Punkt $\vec{x}^{(i)}$ und jedem anderen Punkt $\vec{x}$. Sie wird von zwei Schlüsselsätzen von Hyperparametern gesteuert: $\vec{\theta}$ und $\vec{p}$.

Hyperparameter $\vec{\theta}$ (Theta): Der Aktivitätsparameter

Der Parametervektor $\vec{\theta} = \{\theta_1, \theta_2, ..., \theta_k\}^T$ ist wohl der wichtigste Hyperparameter im Kriging-Modell. Jede Komponente θ_j steuert, wie schnell die Korrelation mit dem Abstand entlang der j-ten Dimension abfällt. Er wird oft als „Aktivitäts"- oder „Breiten"-Parameter bezeichnet.

- Ein **großes** θ_j zeigt an, dass die Funktion sehr empfindlich auf Änderungen in der j-ten Variablen reagiert. Die Korrelation wird sehr schnell abfallen, wenn sich die Punkte in dieser Dimension voneinander entfernen, was zu einer „schmalen" Basisfunktion führt. Dies impliziert, dass die zugrunde liegende Funktion entlang dieser Achse sehr „aktiv" ist oder sich schnell ändert.
- Ein **kleines** θ_j zeigt an, dass die Funktion relativ unempfindlich auf Änderungen in der j-ten Variablen reagiert. Die Korrelation wird langsam abfallen, was zu einer „breiten" Basisfunktion führt, die ihren Einfluss über einen größeren Bereich ausdehnt.

Die Tatsache, dass $\vec{\theta}$ ein Vektor ist – mit einem separaten Wert für jede Eingabedimension – ist ein entscheidendes Merkmal, das dem Kriging immense Leistungsfähigkeit verleiht, insbesondere bei mehrdimensionalen Problemen. Dies ist als **anisotrope Modellierung**

bekannt. Indem die Korrelationslänge für jede Variable unterschiedlich sein kann, kann sich das Modell an Funktionen anpassen, die sich entlang verschiedener Achsen unterschiedlich verhalten. Zum Beispiel könnte eine Funktion sehr schnell auf Temperaturänderungen, aber sehr langsam auf Druckänderungen reagieren. Ein anisotropes Kriging-Modell kann dieses Verhalten erfassen, indem es ein großes θ für die Temperatur und ein kleines θ für den Druck lernt.

Diese Fähigkeit hat eine tiefgreifende Konsequenz: **automatische Relevanzbestimmung**. Während des Modellanpassungsprozesses (den wir in Kapitel diskutieren werden) findet der Optimierungsalgorithmus die $\vec{\theta}$-Werte, die die Daten am besten erklären. Wenn eine bestimmte Eingangsvariable x_j wenig oder keinen Einfluss auf die Ausgabe y hat, wird das Modell einen sehr kleinen Wert für θ_j lernen. Ein kleines θ_j macht den Term $\theta_j|x_j^{(i)} - x_j|^{p_j}$ nahe null, was die Korrelation effektiv unempfindlich gegenüber Änderungen in dieser Dimension macht. Daher kann ein Ingenieur nach der Anpassung des Modells den optimierten $\vec{\theta}$-Vektor inspizieren, um eine Sensitivitätsanalyse durchzuführen. Die Dimensionen mit den größten θ_j-Werten sind die einflussreichsten Treiber der Systemantwort. Dies verwandelt das Surrogatmodell von einem einfachen Black-Box-Approximator in ein Werkzeug zur Generierung wissenschaftlicher und technischer Erkenntnisse. Der in Kapitel besprochene Python-Code, als eindimensionales Beispiel, vereinfacht dies durch die Verwendung eines einzelnen skalaren `theta`, aber das Verständnis seiner Rolle als Vektor ist entscheidend, um den Nutzen des Kriging in realen Anwendungen zu schätzen [@bart23icode].

Hyperparameter $\vec{p}$: Der Glattheitsparameter

Der Parametervektor $\vec{p} = \{p_1, p_2, ..., p_k\}^T$ steuert die Glattheit der Funktion an den Datenpunkten. Sein Wert ist typischerweise auf das Intervall $[1, 2]$ beschränkt [@Forr08a]. Die Wahl von p_j hat tiefgreifende Auswirkungen auf die Form der resultierenden Basisfunktion:

- Wenn $p_j = 2$, ist die resultierende Funktion unendlich differenzierbar, was bedeutet, dass sie sehr glatt ist. Dies ist im bereitgestellten Python-Code der Fall, was durch die Verwendung der `sqeuclidean`-Distanzmetrik (quadrierter Abstand entspricht $p = 2$) implizit ist. Die `sqeuclidean`-Metrik ist auf https://docs.scipy.org/doc/scipy/reference/generated/scipy.spatial.distance.sqeuclidean.html beschrieben.
- Wenn $p_j = 1$, ist die resultierende Funktion stetig, aber an den Datenpunkten nicht differenzierbar, was ihr ein „spitzeres“ oder „stacheligeres“ Aussehen verleiht.

Die Wahl von p spiegelt eine Annahme über die Natur der zugrunde liegenden Funktion wider. Die meisten physikalischen Prozesse sind glatt, was $p = 2$ zu einer gängigen und robusten Wahl macht. Für Funktionen mit bekannten scharfen Merkmalen kann jedoch die Optimierung von p in Richtung 1 eine bessere Anpassung ermöglichen.

Aufbau des Systems: Die Korrelationsmatrizen Ψ und $\vec{\psi}$

Mit dem definierten Korrelationskernel können wir nun die Matrizen konstruieren, die den Kern des Kriging-Systems bilden. Diese Matrizen quantifizieren die Beziehungen zwischen allen Punkten in unserem Problem: den bekannten Trainingspunkten und den neuen Vorhersagepunkten.

Die `Psi`-Matrix (Ψ): Korrelation der Trainingsdaten

Die Matrix Ψ ist die $n \times n$ Korrelationsmatrix der n Trainingsdaten mit sich selbst. Jedes Element Ψ_{ij} ist die Korrelation zwischen dem Trainingspunkt $\vec{x}^{(i)}$ und dem Trainingspunkt $\vec{x}^{(j)}$, berechnet mit der Basisfunktion.

$$\Psi_{ij} = \text{corr}(\vec{x}^{(i)}, \vec{x}^{(j)}) = \exp\left(-\sum_{l=1}^{k} \theta_l |x_l^{(i)} - x_l^{(j)}|^{p_l}\right).$$

Da die Korrelation eines Punktes mit sich selbst perfekt ist, sind die diagonalen Elemente Ψ_{ii} immer gleich 1. Die Matrix ist auch symmetrisch, da der Abstand von Punkt A zu B derselbe ist wie von B zu A.

Code-Analyse (`build_Psi`): Die bereitgestellte Python-Funktion `build_Psi` implementiert diese Berechnung effizient.

```
def build_Psi(X, theta, eps=sqrt(spacing(1))):
    D = squareform(pdist(X, metric='sqeuclidean', out=None, w=theta))
    Psi = exp(-D)
    Psi += multiply(eye(X.shape[0]), eps)
    return Psi
```

1. `D = squareform(pdist(X, metric='sqeuclidean', out=None, w=theta))`: Diese Zeile ist der rechnerische Kern. Die Funktion `scipy.spatial.distance.pdist` berechnet die paarweisen Abstände zwischen allen Zeilen in der Eingabematrix `X_train`.
   - `metric='sqeuclidean'` gibt an, dass der quadrierte euklidische Abstand, $(x_i - x_j)^2$, verwendet werden soll. Dies setzt implizit den Hyperparameter $p = 2$.
   - `w=theta` wendet den Aktivitätsparameter als Gewicht auf den quadrierten Abstand jeder Dimension an und berechnet $\theta_j(x_{ij} - x_{kj})^2$ für jedes Paar von Punkten i, k und jede Dimension j. Für den 1D-Fall im Code ist dies einfach `theta * (x_i - x_j)^2`.
   - `squareform` wandelt dann den von `pdist` zurückgegebenen komprimierten Abstandsvektor in die vollständige, symmetrische $n \times n$ Abstandsmatrix um, die wir D nennen können.
2. `Psi = exp(-D)`: Dies führt eine elementweise Potenzierung des Negativen der Abstandsmatrix durch und vervollständigt die Berechnung des Gauß'schen Kernels.

3. `Psi += multiply(eye(X.shape), eps)`: Diese Zeile addiert eine kleine Konstante `eps` zur Diagonale der Ψ-Matrix. Dies ist der „Nugget"-Term, eine entscheidende Komponente sowohl für die numerische Stabilität als auch für die Rauschmodellierung, die in Kapitel behandelt wird.

Der `psi`-Vektor/Matrix ($\vec{\psi}$): Vorhersage-Trainings-Korrelation

Die Matrix $\vec{\psi}$ ist die $n \times m$ Matrix der Korrelationen zwischen den n bekannten Trainingspunkten und den m neuen Punkten, an denen wir eine Vorhersage machen möchten. Jedes Element ψ_{ij} ist die Korrelation zwischen dem i-ten Trainingspunkt $\vec{x}^{(i)}$ und dem j-ten Vorhersagepunkt $\vec{x}^{(j)}_{pred}$.

$$\psi_{ij} = \text{corr}(\vec{x}^{(i)}, \vec{x}^{(j)}_{pred})$$

Code-Analyse (`build_psi`): Die Funktion `build_psi` berechnet diese Matrix.

```
def build_psi(X_train, x_predict, theta):
    D = cdist(x_predict, X_train, metric='sqeuclidean', out=None, w=theta)
    psi = exp(-D)
    return psi.T
```

1. `D = cdist(x_predict, X_train, metric='sqeuclidean', out=None, w=theta)`: Hier wird `scipy.spatial.distance.cdist` verwendet, siehe https://docs.scipy.org/doc/scipy/reference/generated/scipy.spatial.distance.cdist.html, da wir Abstände zwischen Punkten aus zwei *verschiedenen* Mengen berechnen: den m Vorhersagepunkten in `x_predict` und den n Trainingspunkten in `X_train`. Dies führt zu einer $m \times n$ Matrix gewichteter quadrierter Abstände.
2. `psi = exp(-D)`: Wie zuvor vervollständigt dies die Kernel-Berechnung.
3. `return psi.T`: Die resultierende Matrix wird transponiert, um die Größe $n \times m$ zu haben. Dies ist eine Konvention, um mit der in den Vorhersageformeln in Referenztexten wie @Forr08a dargestellten Matrixalgebra übereinzustimmen.

Modellkalibrierung durch Maximum-Likelihood-Schätzung (MLE)

Sobald die Struktur des Modells durch den Kernel definiert ist, müssen wir die optimalen Werte für seine Hyperparameter, nämlich $\vec{\theta}$ und $\vec{p}$, bestimmen. Der Code in Kapitel umgeht diesen Schritt, indem er `theta = 1.0` fest codiert, aber in jeder praktischen Anwendung müssen diese Parameter aus den Daten gelernt werden. Eine gebräuchliche Methode hierfür ist die **Maximum-Likelihood-Schätzung (MLE)**.

Die Kernidee der MLE besteht darin, die Frage zu beantworten: „Welche Werte der Hyperparameter machen bei unseren beobachteten Daten diese Daten am wahrscheinlichsten?"

Wir finden die Parameter, die die Wahrscheinlichkeit maximieren, die Daten beobachtet zu haben, die wir tatsächlich gesammelt haben.

Die Likelihood-Funktion

Unter der Annahme des Gauß-Prozesses wird die gemeinsame Wahrscheinlichkeit, den Antwortvektor $\vec{y}$ bei gegebenen Parametern zu beobachten, durch die multivariate normale Wahrscheinlichkeitsdichtefunktion beschrieben. Diese Funktion ist unsere Likelihood, L:

$$L(\mu, \sigma^2, \vec{\theta}, \vec{p}|\vec{y}) = \frac{1}{(2\pi\sigma^2)^{n/2}|\Psi|^{1/2}} \exp\left[-\frac{(\vec{y} - \vec{1}\mu)^T \vec{\Psi}^{-1}(\vec{y} - \vec{1}\mu)}{2\sigma^2}\right]. \tag{1}$$

Hier sind μ und σ^2 der globale Mittelwert und die Varianz des Prozesses, und Ψ ist die Korrelationsmatrix, die implizit von $\vec{\theta}$ und $\vec{p}$ abhängt. Beachten Sie, dass $|\Psi|$ die Determinante (also ein reellwertiger Skalar) der Korrelationsmatrix ist.

Die konzentrierte Log-Likelihood

Die direkte Maximierung der Likelihood-Funktion (Gleichung 1) ist schwierig. Aus Gründen der rechnerischen Stabilität und mathematischen Bequemlichkeit arbeiten wir stattdessen mit ihrem natürlichen Logarithmus, der Log-Likelihood (siehe Gleichung (2.29) in @Forr08a):

$$\ln(L) = -\frac{n}{2}\ln(2\pi) - \frac{n}{2}\ln(\sigma^2) - \frac{1}{2}\ln|\Psi| - \frac{(\vec{y} - \vec{1}\mu)^T \Psi^{-1}(\vec{y} - \vec{1}\mu)}{2\sigma^2}. \tag{2}$$

Eine wesentliche Vereinfachung ergibt sich, da wir für jedes gegebene $\vec{\theta}$ und $\vec{p}$ (und damit ein festes Ψ) die optimalen Werte für μ und σ^2 analytisch finden können, indem wir Ableitungen bilden und sie auf null setzen. Die MLE für den Mittelwert ist (siehe Gleichung (2.30) in @Forr08a):

$$\hat{\mu} = \frac{\mathbf{1}^T \Psi^{-1} \vec{y}}{\mathbf{1}^T \Psi^{-1} \mathbf{1}}. \tag{3}$$

Die Berechnung in Gleichung 3 kann als Berechung eines verallgemeinerten gewichteten Durchschnitts der beobachteten Antworten interpretiert werden, wobei die Gewichtung die Korrelationsstruktur berücksichtigt.

Ein ähnlicher Ausdruck existiert für die optimale Varianz, $\hat{\sigma}^2$ (siehe Gleichung (2.31) in @Forr08a):

$$\hat{\sigma}^2 = \frac{(\vec{y} - \mathbf{1}\hat{\mu})^T \Psi^{-1}(\vec{y} - \mathbf{1}\hat{\mu})}{n}.$$

Indem wir diese analytischen Ausdrücke für $\hat{\mu}$ und $\hat{\sigma}^2$ wieder in die Log-Likelihood-Funktion (Gleichung 2) einsetzen, erhalten wir die **konzentrierte Log-Likelihood-Funktion** (siehe Gleichung (2.32) in @Forr08a):

$$\ln(L) \approx -\frac{n}{2}\ln(\hat{\sigma}^2) - \frac{1}{2}\ln|\Psi|.$$

Diese vereinfachte Funktion hängt nun nur noch von den Hyperparametern $\vec{\theta}$ und $\vec{p}$ ab, die in Ψ und $\hat{\sigma}^2$ eingebettet sind.

Numerische Optimierung

Wir können die optimalen $\vec{\theta}$ und $\vec{p}$ nicht analytisch bestimmen. Die konzentrierte Log-Likelihood ist jedoch eine skalare Funktion, die relativ schnell zu berechnen ist. Wir können daher einen numerischen Optimierungsalgorithmus – wie einen genetischen Algorithmus, Nelder-Mead oder Differential Evolution – verwenden, um den Hyperparameterraum zu durchsuchen und die Werte von $\vec{\theta}$ und $\vec{p}$ zu finden, die diese Funktion maximieren. Dieser Suchprozess ist die „Trainings"- oder „Anpassungs"-Phase beim Aufbau eines Kriging-Modells. Sobald die optimalen Hyperparameter gefunden sind, ist das Modell vollständig kalibriert und bereit, Vorhersagen zu treffen.

Implementierung und Vorhersage

Nachdem der theoretische und mathematische Rahmen geschaffen wurde, konzentriert sich dieser Teil auf die praktische Anwendung des Kriging-Modells: wie man es zur Erstellung von Vorhersagen verwendet und welche wesentlichen numerischen Techniken sicherstellen, dass der Prozess sowohl effizient als auch robust ist.

Der Kriging-Prädiktor: Generierung neuer Werte

Das ultimative Ziel beim Aufbau eines Kriging-Modells ist die Vorhersage des Funktionswerts an neuen, unbeobachteten Punkten. Die hierfür verwendete Formel ist als **Bester Linearer Unverzerrter Prädiktor (BLUP)** bekannt. Sie liefert den Mittelwert des posterioren Gauß-Prozesses am Vorhersageort, was unsere beste Schätzung des Funktionswerts ist.

Die Vorhersageformel lautet (siehe Gleichung (2.40) in @Forr08a):

$$\hat{y}(\vec{x}) = \hat{\mu} + \vec{\psi}^T \Psi^{-1}(\vec{y} - \mathbf{1}\hat{\mu}), \tag{4}$$

wobei $\hat{y}(\vec{x})$ die Vorhersage an einem neuen Punkt $\vec{x}$ ist und alle anderen Terme wie im Glossar definiert sind.Im Folgenden beschreiben wir die einzelnen Komponenten dieser Gleichung und ihre Bedeutung:

1. $\hat{\mu}$: Die Vorhersage beginnt mit dem geschätzten globalen Mittelwert des Prozesses. Dies ist unsere Basisvermutung, bevor wir den Einfluss der lokalen Datenpunkte berücksichtigen.
2. $(\vec{y} - \mathbf{1}\hat{\mu})$: Dies ist der Vektor der Residuen. Er stellt die Differenz zwischen jedem beobachteten Datenpunkt und dem globalen Mittelwert dar. Dieser Vektor erfasst die spezifischen, lokalen Informationen, die unsere Trainingsdaten liefern.

3. $\Psi^{-1}(\vec{y} - \mathbf{1}\hat{\mu})$: Dieser Term kann als ein Vektor von Gewichten betrachtet werden, nennen wir ihn $\vec{w}$. Indem wir die Residuen mit der Inversen der Korrelationsmatrix multiplizieren, berechnen wir einen Satz von Gewichten, der die Interkorrelationen zwischen den Trainingspunkten berücksichtigt. Dieser Schritt sagt im Wesentlichen: „Wie viel sollte jedes Residuum beitragen, wenn man bedenkt, dass die Datenpunkte selbst nicht unabhängig sind?“
4. $\vec{\psi}^T\vec{w}$: Die endgültige Vorhersage wird durch eine gewichtete Summe dieser berechneten Gewichte angepasst. Die Gewichte für diese Summe sind die Korrelationen ($\vec{\psi}$) zwischen dem neuen Vorhersagepunkt $\vec{x}$ und jedem der Trainingspunkte. Im Wesentlichen besagt die Formel: „Beginne mit dem globalen Mittelwert und füge dann eine Korrektur basierend auf den beobachteten Residuen hinzu. Der Einfluss jedes Residuums wird durch die Korrelation des neuen Punktes mit dem entsprechenden Trainingspunkt bestimmt.“

Code-Analyse (`mu_hat` und `f_predict`): Der bereitgestellte Python-Code implementiert diesen Vorhersageprozess direkt.

```
U = cholesky(Psi).T
one = np.ones(n).reshape(-1, 1)
mu_hat = (one.T @ solve(U, solve(U.T, y_train))) / (one.T @ solve(U, solve(U.T, one)))
f_predict = mu_hat * np.ones(m).reshape(-1, 1) + psi.T @ solve(U, solve(U.T, y_train - one *
```

- `mu_hat = (one.T @ solve(U, solve(U.T, y_train))) / (one.T @ solve(U, solve(U.T, one)))`: Dies ist eine direkte und numerisch stabile Implementierung der Formel für $\hat{\mu}$, siehe Gleichung 3. Anstatt `Psi_inv` explizit zu berechnen, wird der auf Cholesky basierende Löser verwendet, der im nächsten Abschnitt detailliert wird. Der Ausdruck `solve(U, solve(U.T, y_train))` ist äquivalent zu `Psi_inv @ y_train`.
- `f_predict = mu_hat *... + psi.T @ solve(U, solve(U.T, y_train - one * mu_hat))`: Diese Zeile ist eine direkte Übersetzung der BLUP-Formel aus Gleichung 4. Sie berechnet die Residuen `y_train - one * mu_hat`, multipliziert sie mit `Psi_inv` unter Verwendung des Cholesky-Lösers und berechnet dann das Skalarprodukt mit `psi.T`, bevor das Ergebnis zum Basiswert `mu_hat` addiert wird.

Numerische Best Practices und der Nugget-Effekt

Eine direkte Implementierung der Kriging-Gleichungen unter Verwendung der Standard-Matrixinversion kann sowohl langsam als auch numerisch anfällig sein. Professionelle Implementierungen stützen sich auf spezifische numerische Techniken, um Effizienz und Robustheit zu gewährleisten.

Effiziente Inversion mit Cholesky-Zerlegung

Die Berechnung der Matrixinversen Ψ^{-1} ist der rechenintensivste Schritt sowohl im MLE-Prozess als auch bei der endgültigen Vorhersage. Eine direkte Inversion hat eine Rechenkomplexität von etwa $O(n^3)$. Wenn die Matrix schlecht konditioniert ist (fast singulär), kann die direkte Inversion außerdem zu großen numerischen Fehlern führen.

Ein überlegener Ansatz ist die Verwendung der **Cholesky-Zerlegung**. Diese Methode gilt für symmetrische, positiv definite Matrizen wie Ψ. Sie zerlegt Ψ in das Produkt einer unteren Dreiecksmatrix L und ihrer Transponierten L^T (oder einer oberen Dreiecksmatrix U und ihrer Transponierten U^T), sodass $\Psi = LL^T$. Diese Zerlegung ist schneller als die Inversion, mit einer Komplexität von ungefähr $O(n^3/3)$ [@Forr08a].

Sobald Ψ zerlegt ist, wird das Lösen eines linearen Systems wie $\Psi\vec{w} = \vec{b}$ zu einem zweistufigen Prozess der Lösung zweier viel einfacherer Dreieckssysteme, ein Verfahren, das als Vorwärts- und Rückwärtssubstitution bekannt ist:

1. Löse $L\vec{v} = \vec{b}$ nach $\vec{v}$.
2. Löse $L^T\vec{w} = \vec{v}$ nach $\vec{w}$.

Genau das macht der Python-Code. Die Zeile `U = cholesky(Psi).T` führt die Zerlegung durch (NumPys `cholesky` gibt den unteren Dreiecksfaktor L zurück, also wird er transponiert, um den oberen Dreiecksfaktor U zu erhalten). Anschließend implementieren Ausdrücke wie `solve(U, solve(U.T,...))` die effiziente zweistufige Lösung, ohne jemals die vollständige Inverse von Ψ zu bilden.

Der Nugget: Von numerischer Stabilität zur Rauschmodellierung

Die Cholesky-Zerlegung funktioniert nur, wenn die Matrix Ψ streng positiv definit ist. Ein Problem tritt auf, wenn zwei Trainingspunkte $\vec{x}^{(i)}$ und $\vec{x}^{(j)}$ sehr nahe beieinander liegen. In diesem Fall wird ihre Korrelation nahe 1 sein, was die entsprechenden Zeilen und Spalten in Ψ nahezu identisch macht. Dies führt dazu, dass die Matrix schlecht konditioniert oder fast singulär wird, was zum Scheitern der Cholesky-Zerlegung führen kann.

Dieses Problem wird gelöst, indem ein kleiner positiver Wert zur Diagonale der Korrelationsmatrix addiert wird: $\Psi_{new} = \Psi + \lambda I$, wobei I die Identitätsmatrix ist. Diese kleine Addition, oft als **Nugget** bezeichnet, stellt sicher, dass die Matrix gut konditioniert und invertierbar bleibt. Im bereitgestellten Code dient die Variable `eps` diesem Zweck.

Obwohl es als numerischer „Hack“ beginnen mag, hat dieser Nugget-Term eine tiefgreifende und starke statistische Interpretation: Er modelliert Rauschen in den Daten. Dies führt zu zwei unterschiedlichen Arten von Kriging-Modellen:

1. **Interpolierendes Kriging ($\lambda \approx 0$):** Wenn der Nugget null oder sehr klein ist (wie `eps` im Code), wird das Modell gezwungen, exakt durch jeden Trainingsdatenpunkt zu verlaufen. Dies ist für deterministische Computerexperimente geeignet, bei denen die Ausgabe rauschfrei ist.
2. **Regressives Kriging ($\lambda > 0$):** Wenn bekannt ist, dass die Daten verrauscht sind (z. B. aus physikalischen Experimenten oder stochastischen Simulationen), würde das Erzwingen der Interpolation jedes Punktes dazu führen, dass das Modell das Rauschen anpasst, was zu einer übermäßig komplexen und „zappeligen“ Oberfläche führt, die schlecht generalisiert. Durch Hinzufügen eines größeren Nugget-Terms λ zur Diagonale teilen wir dem Modell explizit mit, dass es Varianz (Rauschen) in den Beobachtungen gibt. Das Modell ist nicht mehr verpflichtet, exakt durch die Datenpunkte zu verlaufen. Stattdessen wird es eine glattere Regressionskurve erstellen, die den zugrunde liegenden Trend erfasst und gleichzeitig das Rauschen herausfiltert. Die Größe von λ kann als weiterer zu optimierender Hyperparameter behandelt werden, der die Varianz des Rauschens darstellt.

Dieselbe mathematische Operation – das Hinzufügen eines Wertes zur Diagonale von Ψ – dient somit einem doppelten Zweck. Ein winziges, festes `eps` ist eine pragmatische Lösung für die numerische Stabilität. Ein größeres, potenziell optimiertes λ ist ein formaler statistischer Parameter, der das Verhalten des Modells grundlegend von einem exakten Interpolator zu einem rauschfilternden Regressor ändert.

Eine vollständige exemplarische Vorgehensweise: Kriging der Sinusfunktion

Dieser letzte Teil fasst die gesamte vorangegangene Theorie zusammen, indem er sie auf das bereitgestellte Python-Codebeispiel aus dem Hyperparameter Tuning Cookbook The Complete Python Code for the Example anwendet. Wir werden das Skript Schritt für Schritt durchgehen und die Eingaben, Prozesse und Ausgaben in jeder Phase interpretieren, um eine konkrete Veranschaulichung des Kriging in Aktion zu geben.

Schritt-für-Schritt-Codeausführung und Interpretation

Das Beispiel zielt darauf ab, die Funktion $y = \sin(x)$ unter Verwendung einer kleinen Anzahl von Stichprobenpunkten zu modellieren. Dies ist ein klassisches „Spielzeugproblem“, das nützlich ist, um zu visualisieren, wie sich das Modell verhält.

Schritt 1: Datengenerierung

```
n = 8
X_train = np.linspace(0, 2 * np.pi, n, endpoint=False).reshape(-1, 1)
y_train = np.sin(X_train)
```

Hier generiert das Skript die Trainingsdaten.

- `n = 8`: Wir entscheiden uns, die Funktion an acht verschiedenen Stellen abzutasten.
- `X_train`: Dies erstellt ein Array von acht gleichmäßig verteilten Punkten im Intervall [0, 2π], die als Trainingspunkte dienen.
- `y_train = np.sin(X_train)`: Dies berechnet die Sinuswerte an den Trainingspunkten. Das Ergebnis ist ein 8×1 Vektor, der die beobachteten Antworten darstellt.

Schritt 2: Definition der Korrelationsmatrix (Ψ)

```
theta = np.array([1.0])
Psi = build_Psi(X_train, theta)
```

Dieser Schritt berechnet die 8×8 Korrelationsmatrix Ψ für die Trainingsdaten.

- `theta = np.array([1.0])`: Der Aktivitätshyperparameter θ wird auf 1.0 gesetzt. In einer realen Anwendung würde dieser Wert durch MLE gefunden, aber hier wird er zur Vereinfachung festgesetzt.
- `Psi = build_Psi(X_train, theta)`: Die Funktion `build_Psi` wird aufgerufen. Sie berechnet den gewichteten quadrierten Abstand zwischen jedem Paar von Punkten in `X_train` und wendet dann den exponentiellen Kernel an. Die resultierende `Psi`-Matrix quantifiziert die angenommene Korrelation zwischen all unseren bekannten Datenpunkten. Die diagonalen Elemente sind 1 (plus ein winziges `eps`), und die außerdiagonalen Werte nehmen ab, wenn der Abstand zwischen den entsprechenden Punkten zunimmt.

Schritt 3: Definition der Vorhersagepunkte

```
m = 100
x_predict = np.linspace(0, 2 * np.pi, m, endpoint=True).reshape(-1, 1)
```

Wir definieren nun die Orte, an denen wir neue Vorhersagen machen wollen. Wir erstellen ein dichtes Gitter von `m = 100` Punkten, das das gesamte Intervall [0, 2π] abdeckt. Dies sind die Punkte, an denen wir unser Surrogatmodell auswerten werden, um eine glatte Kurve zu erzeugen.

Schritt 4: Berechnung der Vorhersagekorrelation ($\vec{\psi}$)

```
psi = build_psi(X_train, x_predict, theta)
```

Dieser Schritt berechnet die 8×100 Korrelationsmatrix $\vec{\psi}$. Die Funktion `build_psi` berechnet die Korrelation zwischen jedem der acht Trainingspunkte und jedem der 100 Vorhersagepunkte. Jede Spalte der resultierenden `psi`-Matrix entspricht einem Vorhersagepunkt und enthält seine acht Korrelationswerte mit dem Trainingssatz. Diese Matrix verbindet die neuen, unbekannten Orte mit unserer bestehenden Wissensbasis.

Schritt 5: Berechnung der Vorhersage

```
U = cholesky(Psi).T
one = np.ones(n).reshape(-1, 1)
mu_hat = (one.T @ solve(U, solve(U.T, y_train))) / (one.T @ solve(U, solve(U.T, one)))
f_predict = mu_hat * np.ones(m).reshape(-1, 1) + psi.T @ solve(U, solve(U.T, y_train - one *
```

Dies ist der entscheidende Schritt, in dem die tatsächlichen Vorhersagen gemacht werden.

1. `U = cholesky(Psi).T`: Die numerisch entscheidende Cholesky-Zerlegung von Ψ wird durchgeführt.
2. `mu_hat =...`: Die Maximum-Likelihood-Schätzung für den globalen Mittelwert μ wird unter Verwendung des numerisch stabilen Cholesky-Lösers berechnet. Für diese spezifischen symmetrischen Daten wird `mu_hat` nahe null sein.
3. `f_predict =...`: Die BLUP-Formel wird implementiert. Sie berechnet die Residuen (`y_train - one * mu_hat`), findet die gewichteten Residuen unter Verwendung des Cholesky-Lösers (`solve(U, solve(U.T,...))`) und berechnet dann die endgültige Vorhersage durch eine gewichtete Summe basierend auf den Vorhersagekorrelationen `psi.T`. Das Ergebnis, `f_predict`, ist ein 100×1 Vektor, der die vorhergesagten Sinuswerte an jedem der `x_predict`-Orte enthält.

Schritt 6: Visualisierung

Der letzte Codeblock stellt die Ergebnisse grafisch dar.

- **Messungen (Blaue Punkte):** Die ursprünglichen 8 Datenpunkte werden dargestellt.
- **Wahre Sinusfunktion (Graue gestrichelte Linie):** Die tatsächliche $\sin(x)$-Funktion wird als Referenz dargestellt.
- **Kriging-Vorhersage (Orange Linie):** Die vorhergesagten Werte `f_predict` werden gegen `x_predict` aufgetragen.

Die Grafik zeigt die Schlüsseleigenschaften des Kriging-Modells. Die orangefarbene Linie verläuft *exakt* durch jeden der blauen Punkte und demonstriert damit ihre **interpolierende** Natur (da `eps` sehr klein war). Wichtiger noch, zwischen den Stichprobenpunkten liefert die Vorhersage eine glatte und bemerkenswert genaue Annäherung an die wahre zugrunde liegende Sinuskurve, obwohl das Modell in diesen Bereichen keine anderen Informationen über die Funktion hatte als die acht Stichprobenpunkte und die angenommene Korrelationsstruktur.

Fazit und Ausblick

Wir begannen damit, das Kriging als eine anspruchsvolle Form der Modellierung mit radialen Basisfunktionen einzuordnen und seine Kernphilosophie zu übernehmen, deterministische Funktionen als Realisierungen eines stochastischen Prozesses zu behandeln. Anschließend haben wir seine Architektur zerlegt: den leistungsstarken Korrelationskernel mit seinen Aktivitäts- (θ) und Glattheits- (p) Hyperparametern, die Konstruktion der Korrelationsmatrizen (Ψ und $\vec{\psi}$) und den Ansatz der Maximum-Likelihood-Schätzung zur Modellkalibrierung. Schließlich haben wir die numerischen Best Practices wie die Cholesky-Zerlegung untersucht und die doppelte Rolle des Nugget-Terms für die numerische Stabilität und die statistische Rauschmodellierung besprochen. Die schrittweise exemplarische Vorgehensweise des Sinusbeispiels lieferte eine konkrete Demonstration dieser Konzepte und zeigte, wie eine kleine Menge von Datenpunkten verwendet werden kann, um ein genaues, interpolierendes Modell einer unbekannten Funktion zu erzeugen.

Für den angehenden Praktiker ist dies nur der Anfang. Die Welt des Kriging und der Gauß-Prozess-Regression ist reich an fortgeschrittenen Techniken, die auf diesen Grundlagen aufbauen.

- **Fehlerschätzungen und sequentielles Design:** Ein wesentliches Merkmal, das im Beispielcode nicht untersucht wurde, ist, dass das Kriging nicht nur eine mittlere Vorhersage, sondern auch eine **Varianz** an jedem Punkt liefert, die die Unsicherheit des Modells quantifiziert. Diese Varianz ist in Regionen weit entfernt von Datenpunkten hoch und in deren Nähe niedrig. Diese Fehlerschätzung ist die Grundlage für **aktives Lernen** oder **sequentielles Design**, bei dem Infill-Kriterien wie die **erwartete Verbesserung (EI)** verwendet werden, um den nächsten zu beprobenden Punkt intelligent auszuwählen, wobei ein Gleichgewicht zwischen der Notwendigkeit, vielversprechende Regionen auszunutzen (niedriger vorhergesagter Wert) und der Notwendigkeit, unsichere Regionen zu erkunden (hohe Varianz), hergestellt wird.
- **Gradienten-erweitertes Kriging:** In vielen modernen Simulationsumgebungen ist es möglich, nicht nur den Funktionswert, sondern auch seine Gradienten (Ableitungen) zu geringen zusätzlichen Kosten zu erhalten. Das **Gradienten-erweiterte Kriging** integriert diese Gradienteninformationen in das Modell, was seine Genauigkeit drastisch verbessert und den Aufbau hochpräziser Modelle mit sehr wenigen Stichprobenpunkten ermöglicht.

- **Multi-Fidelity-Modellierung (Co-Kriging):** Oft haben Ingenieure Zugang zu mehreren Informationsquellen mit unterschiedlicher Genauigkeit und Kosten – zum Beispiel ein schnelles, aber ungenaues analytisches Modell und eine langsame, aber hochpräzise CFD-Simulation. **Co-Kriging** ist ein Rahmenwerk, das diese unterschiedlichen Datenquellen verschmilzt, indem es die reichlich vorhandenen billigen Daten verwendet, um einen Basistrend zu etablieren, und die spärlichen teuren Daten, um ihn zu korrigieren, was zu einem Modell führt, das genauer ist als eines, das nur aus einer der Datenquellen allein erstellt wurde.

Zusatzmaterialien

Interaktive Webseite

- Eine interaktive Webseite zum Thema **Kriging** ist hier zu finden: Kriging Interaktiv.
- Eine interaktive Webseite zum Thema **MLE** ist hier zu finden: MLE Interaktiv.

Jupyter-Notebook

- Das Jupyter-Notebook für dieses Lernmodul ist auf GitHub im Hyperparameter-Tuning-Cookbook Repository verfügbar.