VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra aplikované matematiky

Po částech lineární regrese

Segmented Regression

2018

Martin Koběrský

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra aplikované matematiky

Zadání diplomové práce

Student: **Bc. Martin Koběrský**

Studijní program: N2647 Informační a komunikační technologie

Studijní obor: 1103T031 Výpočetní matematika

Téma: Po částech lineární regrese
Piecewise linear regression

Jazyk vypracování: čeština

Zásady pro vypracování:

Lineární regresní modely patří mezi nejčastěji používané statistické modely. V některých aplikacích ale nelze závislost náhodných veličin jednoduše modelovat v celém oboru vysvětlující veličiny. Místo toho může být vhodné obor hodnot vysvětlující veličiny rozdělit na disjunktní intervaly a v nich závislost vysvětlované veličiny modelovat různými funkcemi, velmi často lineárními. Pokud meze intervalů nelze předem určit, stávají se dalšími neznámými parametry modelu. Příkladem takové úlohy může být modelování závislosti životnosti materiálu při cyklickém namáhání na amplitudě zátěžového cyklu. Závislost se mění v tzv. oblastech vysokocyklové únavy, nízkocyklové únavy a oblasti bezpečného namáhání.

S využitím bayesovské statistiky lze takovou úlohu formulovat poměrně přímočaře. K samotnému odhadu parametrů je ale potřeba numerického řešení, typicky pomocí Markov Chain Monte Carlo metod (MCMC). K tomuto účelu lze využít některou z knihoven pro MCMC výpočty, např. PyMC. Jako u většiny podobných úloh i zde lze očekávat, že získání dobrého odhadu bude podmíněno vhodnou volbou apriorního rozdělení reprezentujícího apriorní informaci.

Pokyny pro vypracování:

Seznamte se s principy metod bayesovské statistiky a základy MCMC algoritmů. Na souborech simulovaných dat ověřte vliv parametrizace modelu, apriorního rozdělení a počtu dat na tvar aposteriorního rozdělení a pokuste se výsledky interpretovat. Po dohodě se školitelem využijte po částech lineární regresní model pro analýzu reálných dat.

Seznam doporučené odborné literatury:

Robert, Christian. The Bayesian choice: from decision-theoretic foundations to computational implementation. Springer Science & Business Media, 2007.

Robert, Christian, and George Casella. Monte Carlo statistical methods. Springer Science & Business Media, 2013.

PyMC User's Guide: https://pymc-devs.github.io/pymc/

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **Ing. Jan Kracík, Ph.D.**

Datum zadání: 01.09.2017

Datum odevzdání: 30.04.2018

doc. RNDr. Jiří Bouchala, Ph.D.	prof. Ing. Pavel Brandštetter, CSc.
vedoucí katedry	*děkan fakulty*

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 30. dubna 2018 .....

Souhlasím se zveřejněním této diplomové práce dle požadavků čl. 26, odst. 9 Studijního a zkušebního řádu pro studium v magisterských programech VŠB-TU Ostrava.

V Ostravě 30. dubna 2018 .....

Rád bych poděkoval panu Kracíkovi za shovívavost a trpělivost a svojí mamince za zkontrolování gramtických chyb. Pokud i přesto nějaké chyby zůstaly, chyba je jen na mojí straně. Děkuji.

Abstrakt

Práce se zabývá modelem po částech linearní regrese. Obsahuje stručný úvod do Bayesovské statistiky, pomcí níž zavádíme model pro po částech lineární regresi. Protože se s aposteriorním rozdělením dáneho modelu nedá analyticky pracovat, používáme pro jeho aproximaci algoritmus RJMCMC. K tomu se postupně dostáváme přes klasické MCMC metody. Na závěr práce prezentujeme výsledky dosažené za pomocí algoritmu RJMCMC, jehož implementace je součásti této práce a byla provedena v programovacím jazyce Python.

Klíčová slova: diplomová práce, Python, MCMC, RJMCM, Po částech lineární regrese, lineární regrese, Bayesovská statistika, Theano, SciPy

Abstract

This work is about segmented linear regression. It contains a brief introduction to Bayesian statistics, which we use to define a model for segmented linear regression. Posterior distribution for the model doesn't have a closed form, so we use RJMCMC algorithm to aproximate the distribution. Before introducing RJMCMC we give an overview of classical MCMC algorithms. At the end of the thesis we show examples of how RJMCMC works. We implemented the RJMCMC algorithm as part of the thesis, for that we used the programming language Python.

Key Words: master's thesis, Python, MCMC, RJMCMC, Piecewise linear regression, Segmented regression, Bayesian statistics, Theano, Scipy

Obsah

Seznam obrázků

Seznam výpisů zdrojového kódu

1 Úvod

Statistické problémy, kde jedna z věcí, kterou hledáme je počet parametrů, se často objevují ve statistickém modelování. Objevují se v novějších problémech jako je rozpoznávaní obrazu, ale také při tradičnějších statických úlohách, jako jsou směsi normálních rozdělení a nebo v po částech lineární regresi.

Druhou zmiňovanou úlohou se budeme zabývat my v této práci. Stručně jde o to najít po částech lineární spojitou funkcí, která co nejlépe vysvětluje zadané data. K namodelování problému využijeme Bayesovskou statistiku, pomocí které dokážeme poměrně elegantně popsat daný model. Její hlavní výhodou při řešení tohoto problému je to, že přirozeně znevýhodňuje modely s vyšší dimenzí [6], které ale nevysvětlují data o mnoho líp než jednodušší modely. V rámci Bayesovské statistiky tedy sestavíme aposteriorní rozdělení na parametrech modelu podmíněně závislé na datech. Pomocí něhož můžeme učinit závěry o daných parametrech.

Protože toto rozdělení je příliš složité na to, abychom s ním mohli pracovat analyticky, budeme se muset uchýlit k aproximaci. Pro tento účel se většinou používají algoritmy Markov Chain Monte Carlo, které ale nestačí na úkoly, kde počet neznámých je jeden z parametrů modelu. Proto využijeme algoritmus Reversible Jump Markov Chain Monte Carlo, který zobecňuje algoritmy MCMC i pro modely s proměnnou dimenzí.

2 Bayesovská statistika

Statistika obecně se dá rozdělit na dva přístupy a to na přístup frekventistický a Bayesovský. Rozdíl mezi nimi je v tom, jak se staví k parametrům daného statistické modelu. Ve statistice frekventistické považujeme tyto parametry za nějaké konkrétní, ale neznáme hodnoty. Zatímco ve statistice Bayesovské považujeme tyto parametry za náhodné veličiny, které mají nějaké pravděpodobnostní rozdělení [6]. Výhodou Bayesovského přístupu je, že nám umožňuje přidat nějaké naše přesvědčení nebo předchozí znalost o parametrech do modelu a tím zpřesnit výsledky.

Jak tedy vypadá Bayesovský model? Jedna jeho část je podmíněné pravděpodobnostní rozdělení náhodné veličiny x, jež závisí na parametru θ

$$f(x|\theta). \tag{1}$$

Další jeho částí je takzvané apriorní rozdělení, což je pravděpodobnostní rozdělení parametru θ. Tímto rozdělením můžeme do modelu uvést nějaké předem známe informace nebo omezení.

$$f(\theta) \tag{2}$$

A toto podle známého Bayesova vzorce můžeme napsat jako

$$f(\theta|x) = \frac{f(x|\theta)f(\theta)}{f(x)}, \tag{3}$$

kde $f(x)$ lze chápat jako normalizační konstantu, protože nezávisí na θ a je možné ji získat vyintegrováním čitatele v 3 přes θ.

$$f(x) = \int f(x|\theta)f(\theta)d\theta. \tag{4}$$

Rozdělení $f(\theta|x)$ se říká aposteriorní a shrnuje všechno co víme o parametru θ, tedy apriorní informaci a informaci z dat.

Často bývá výpočet $f(x)$ analyticky nemožný a proto se používají MCMC metody, u kterých není potřeba znát normativní konstantu, ty si ukážeme v dalších kapitolách. Obvykle se také používá značení

$$f(\theta|x) \propto f(x|\theta)f(\theta), \tag{5}$$

které znamená, že $f(\theta|x)$ se rovná $f(x|\theta)f(\theta)$ až na konstantu nezávislou na θ.

3 Model

Modelem bude po částech lineární regrese, čili

$$y_i = f(x_i) + \epsilon_i \tag{6}$$

kde $f(x_i)$ je po částech lineární spojitá funkce a e_i jsou nezávislé identicky distribuované náhodné veličiny s rozdělením $\mathcal{N}(0, \sigma^2)$. Pokud bychom věděli, jak intervaly vypadají, tak můžeme použít metodu nejmenších čtverců k tomu, abychom dostali přímku pro každý interval. V našem případě však nevíme nejen to, jak intervaly vypadají, ale ani to kolik jich je. Zatím si tedy ukážeme, jak konkrétně vypadá aposteriorní rozdělení našeho modelu a v dalších kapitolách budeme řešit to, jak toto rozdělení aproximovat. Pro názornost začneme modelem beze zlomu, čili klasickou Bayesovkou lineární regresí a poté se dostaneme k obecnému modelu.

3.1 Model bez zlomu

Jak už bylo řečeno, zde se podíváme na model lineární bayesovské regrese. Určíme aposteriorní rozdělení parametrů bayesovské lineární regrese s konjugovaným apriorním rozdělením. Jen pro připomenutí to, že je apriorní rozdělení konjugované vzhledem k pravděpodobnostnímu rozdělení znamená to, že aposteriorní rozdělení bude mít stejnou formu jako apriorní rozdělení [6].

Vycházíme tedy ze standardního pravděpodobnostního modelu lineární regrese:

$$y_i = a + bx_i + \epsilon_i \tag{7}$$

kde a, b jsou parametry přímky a y_i, x_i jsou jednotlivá pozorovaná data. A parametr ϵ představuje náhodnou chybu s rozdělením:

$$\epsilon_i \sim \mathcal{N}(0, \sigma^2),$$

kde σ^2 je neznáme a ϵ_i jsou navzájem nezávislé. Z tohoto můžeme určit rozděleni pro y_i tím, že vzorec (7) vlastně říká y_i se rovná normálně rozdělené náhodné veličině, k jejíž střední hodnotě přičítáme $a + bx_i$ tudíž:

$$y_i \sim \mathcal{N}(a + bx_i, \sigma^2). \tag{8}$$

Je vhodné dodat, že jako vysvětlovaná veličina zde vystupuje y_i a x_i vystupuje jako parametr, který je ale známý, proto nepíšeme, že y_i je podmíněně závislé na x_i. Formuli 8 můžeme také psát jako:

$$f(y_i|a, b, \sigma^2) = \mathcal{N}(y_i|a + bx_i, \sigma^2) \tag{9}$$

Použitím $\mathcal{N}(h|a+bx_i,\sigma^2)$ myslíme pravděpodobnostní hustotu normálního rozdělení se střední hodnotou $a+bx_i$ a rozptylem σ^2.

Takto jsme získali pravděpodobnostní rozdělení y_i. Pro doplnění Bayesovského modelu potřebujeme určit apriorní rozdělení. Už jsme řekli, že apriorní rozdělení by mělo být konjugované vzhledem k pravděpodobnostnímu rozdělení. Apriorní rozdělení v tomto případě má mít formu

$$f(a,b,\sigma^2) = f(\sigma^2)f(a,b|\sigma^2), \tag{10}$$

kde $f(\sigma^2)$ má inverzní gamma rozdělení a $f(a,b|\sigma^2)$ má normální rozdělení. Parametry volíme nějakým vhodným způsobem, pokud máme nějakou představu o oblasti, kde se parametry přímky mohou nacházet, tak volíme normální distribuci se středem v dané oblasti a s vhodně nastaveným rozptylem, tak ať pokryjeme danou oblast. Pokud nevíme nic, tak volíme normální rozdělení se středem třeba v bodě nula, ale s velkým rozptylem. Obdobně pro rozptyl:

$$\sigma^2 \sim Inv - Gamma(\alpha,\beta) \tag{11}$$

$$a,b|\sigma^2 \sim \mathcal{N}((\mu_1,\mu_2),\sigma^2\Lambda_0^{-1}) \tag{12}$$

Vynásobením modelu a apriorního rozdělení dostáváme aposteriorní rozdělení. A protože jsme použili konjugované apriorní rozdělení, tak výsledné aposteriorní rozdělení je takovéto:

$$f(a,b,\sigma^2|y) = f(a,b|\sigma^2,y)f(\sigma^2|y) \tag{13}$$

Pokud bychom použili nekonjugované apriorní rozdělení nebude aposteriorní rozdělení v uzavřené formě a tudíž není možné obecně získat analytické řešení, v tomto případě se dají použít MCMC metody, které dokážou aproximovat navzorkováním.

3.2 Model s více zlomy

Nyní se tedy dostáváme k tomu jak bude vypadat model s více zlomy. V modelu beze zlomů jsme jako parametry použili parametry z rovnice přímky. Pro více zlomů se nám jeví výhodnější model parametrizovat pomocí (samozřejmě) rozptylu, ale místo použití směrnic daných přímek, použijeme několik bodů, které budou určovat souřadnice zlomů po částech lineární funkci. Počet bodů bude záviset na počtu zlomů, takže pokud budeme mít 1 zlom, použijeme body 3, pokud 2 zlomy, použijeme body 4 atd. Takže parametr vypadá následovně:

$$\theta_k = (\sigma^2, s_1, h_1, s_2, h_2, ..., s_{k+1}h_{k+1}), \tag{14}$$

kde σ^2 je rozptyl a pro všechna i body s_i určují x-ové souřadnice bodů určujících přímku a h_i určují y-ové souřadnice. k je index modelu a zároveň počet zlomů. Body (s_1,h_1) a (s_{k+1},h_{k+1}) určují první, respektive poslední bod definující funkci.

Pro tento model už nenajdeme konjugované apriorní rozdělení, což má za následek to že, aposteriorní rozdělení už nebude mít uzavřenou formu a my nebudeme schopni analyticky spočítat parametry nového modelu. Ač se toto může na první pohled zdát značně mrzuté nemusíme zoufat, v dalších kapitolách si ukážeme jak tento problém vyřešit. Prozatím se tedy zaměříme na to, jak náš model bude vypadat. Výhodou je, že už se teď nemusíme ohlížet na nějaké značně restriktivní konjugované apriorní rozdělení, ale můžeme si apriorní rozdělení nastavit takřka, jak chceme.

První si zadefinujeme apriorní rozdělení. Začneme u x-ových souřadnic bodů určujících průběh naší po částech lineární spojité funkce. Od těch budeme chtít hlavně to ať zachovají pořadí. V našem modelu je funkce určena postupně několika body, proto nedává smysl, aby druhý bod byl před prvním nebo aby se jinak otáčelo pořadí. To tedy zavedeme takto

$$f_1(\theta_k) \propto \begin{cases} 1, pokud\ s_1 < s_2 < ... < s_{k+1}, \\ 0, jindy \end{cases} \tag{15}$$

zde není úplně správné použít 1 pro případ, kdy je zachováno pořadí. Je to proto, že tato konstanta by měla být správně normalizována, tak ať integrál přes aposteriorní rozdělení je roven jedné. Proč nám to nevadí se ukáže v další kapitole.

Apriorní rozdělení na y-ových souřadnicích by mělo být co nejvíc neinformativní, protože předpokládáme, že o těchto souřadnicích nic nevíme. My jsme v tomto případě zvolili normální rozdělení s nějakým velkým rozptylem ř pro všechny h:

$$f_2(\theta_k) = \prod_{i=1}^{k+1} \mathcal{N}(h_i|0, \text{ř}). \tag{16}$$

Pro rozptyl použijeme normální rozdělení s tím, že rozptyly menší nebo rovné nule mají nulovou pravděpodobnost:

$$f_3(\theta_k) = \begin{cases} \mathcal{N}(\sigma^2|0, 3), pokud\ \sigma^2 > 0, \\ 0, jindy \end{cases} . \tag{17}$$

Celé apriorní rozdělení je tedy jen pro úplnost:

$$f(\theta_k) = f_1(\theta_k) f_2(\theta_k) f_3(\theta_k). \tag{18}$$

Pravděpodobnostní rozdělení bude podobné jako v modelu bez zlomů. Rozdíl je, že zde je přímka rozdělena na obecně k částí, takže je třeba se vzít v potaz to kde leží bod pro který počítáme pravděpodobnost. Nejdříve si zavedu pomocnou funkci, která nám udá pravděpodobnost pro jeden bod, přes jeden interval. Tu potom využiji v obecnější funkci, která už jde přes

všechny možné intervaly. Takže nějak takto:

$$\hat{f}(y_i|\sigma^2, s_1, h_1, s_2, h_2) = \mathcal{N}(y - [(x - s_1)\frac{h_2 - h_1}{s_2 - s_1} + h_1], \sigma^2), \tag{19}$$

$$f(y|\theta_k) = \sum_{i=0}^{k} I(s_j < X_i < s_{j+1})\hat{f}(y_i|s_j, h_j, s_{j+1}, h_{j+1}). \tag{20}$$

Nyní máme pravděpodobnostní rozdělení a apriorní rozdělení, takže jejich vynásobením a průchodem přes všechny vzorky získáváme aposteriorní rozdělení až na multiplikativní konstantu

$$f(\theta_k|y) \propto \prod_{i=0}^{n} f(y_i|\theta_k)f(\theta_k). \tag{21}$$

3.3 Model s více zlomy a různým počtem dimenzí

V předešlé sekci jsme popsali model s nějakým počtem zlomů. V naší úloze, ale nakonec chceme najít, který z takovýchto modelů je pro naše data nejlepší. Proto nakonec potřebujeme ještě obecnější model a to takový, který dokáže shrnout modely s různým počtem zlomů. Parametr bude v takovém modelu vypadat následovně

$$\theta = \bigcup_{k=0}^{q} (k \times \theta_k), \tag{22}$$

kde $q \in \mathcal{N}$ a omezuje mezi kolika modely můžeme vybírat. Je to technické omezení, tak ať nemusíme pracovat s parametrem nekonečné dimenze. Nyní si zadefinujeme aposteriorní rozdělení, tedy:

$$f(k, \theta) = f(\theta|k)f(k). \tag{23}$$

Pro k volíme takové rozdělení, které favorizuje jednodušší modely oproti složitějším v našem případě volíme geometrické rozdělení s parametrem 0.2:

$$k \sim G(0.2). \tag{24}$$

A pro podmíněné rozdělení θ za k využijeme, apriorní rozdělení, které jsme zavedli v minulé kapitole:

$$f(\theta|k) = \sum_{l=0}^{q} \delta_{l,k} f(\theta_l), \tag{25}$$

čímž myslíme, že počítáme jen s apriorním rozdělením modelu v kterém se právě nacházíme.

Pro pravděpodobnostní rozdělení opět využijeme rozdělení, které jsme si zadefinovali v před-

cházející kapitole:

$$f(y|k,\theta) = \sum_{l=0}^{q} \delta_{l,k} f(y|\theta_l). \tag{26}$$

Jako aposteriorní rozdělení tedy máme:

$$f(k,\theta|y) = f(y|k,\theta) f(k,\theta). \tag{27}$$

4 MCMC

Aposteriorní distribuce mají často takový tvar, že je jakýkoli analytický výpočet nemožný. V takový moment se buď musíme spokojit s jednodušším modelem, jehož aposteriorní distribuce umožňuje analytický výpočet a nebo se musíme uchýlit k numerickým aproximacím dané distribuce. MCMC je skupina algoritmů, která dokáže vzorkovat z dané distribuce. Ze získaných vzorků potom můžeme zjistit například bayesovký interval spolehlivosti jen tím, že si ze vzorků spočítáme kvantily o které máme zájem. Dále se získané vzorky dají využít pro Monte Carlo integraci nebo různé optimalizační problémy.

V této kapitole si nejprve ukážeme jak funguje MC integrace a poté přejdeme k Markovským řetězcům, které jsou základem MCMC algoritmů. Nakonec popíšeme nejrozšířenější MCMC algoritmus a to Metropolis-Hastings.

4.1 Monte Carlo Integrace

V této části budeme řešit obecný problém jak vyřešit integrál ve formě:

$$I = \int_{\mathcal{X}} f(x)p(x)dx, \tag{28}$$

kde $\mathcal{X}$ je nějaká množina přes kterou chceme integrovat. Stejným způsobem je definována střední hodnota náhodné veličiny X $E[h(X)] = I$. Nyní mějme výběr $(X_1, ..., X_m)$ vygenerovaný z hustoty p, poté je přirozené aproximovat 28 průměrem:

$$f_m = \frac{1}{m}\sum_{j=1}^{m} f(x_j) \tag{29}$$

který ze zákona velkých čísel konverguje skoro jistě k $E[h(X)]$. Chceme-li tedy počítat nějaký integrál, který nepochází ze statistické úlohy, je třeba si zvolit návrhovou distribuci z které budeme vzorkovat. Jako nejjednodušší se jeví uniformní distribuce, neboť její použití je snadné a vede jen k přenásobení integrálu konstantou. Toto nemusí především ve vyšších dimenzích být vhodné, protože při procházení dané funkce uniformě můžeme narazit na velmi málo nenulových míst nebo míst, která špatně charakterizují funkce a proto se dostaneme ke správnému výsledku až s velkým počtem vzorků. Tento problém můžeme vyřešit buďto použitím jiné návrhové distribuce, takže za $f(x)$ volíme nějakou $\Phi(x)$ a za $g)(x) = \frac{g(x)}{\Phi(x)}$ tato metoda se nazývá importance sampling [7]. A nebo budeme rovnou vzorkovat z funkce f pomocí nějakého MCMC algoritmu a jen vzorky zprůměrujeme.

Navíc víme, že pokud je rozptyl f(x) konečný, pak rozptyl našeho odhadu je roven $\frac{\sigma_f^2}{N}$ a taky víme, že rozdělení chyby konverguje k normálnímu rozdělení s nulovou střední hodnotou a rozptylem funkce f [4]

$$\sqrt{N})(I_N(f) - I(f)) \to \mathcal{N}(0, \sigma_f^2). \tag{30}$$

4.2 Markovský řetězec

Markovův řetězec je posloupnost náhodných veličin, s pravděpodobností přechodu závisející na konkretním stavu ve kterém se řetězec nachází. Proto se řetězec definuje přechodovým jádrem, což je funkce určující přechody.

Definice 1 *Přechodové jádro je funkce K definovaná na $\mathcal{X} \times \mathcal{B}(\mathcal{X})$ taková, že*

- $\forall x \in \mathcal{X}$, $K(x, \dot{)}$ *je pravděpodobnostní míra,*
- $\forall A \in \mathcal{B}(\mathcal{X})$, $K(;A)$ *je měřitelná.*

Když je $\mathcal{X}$ diskrétní, přechodové jádro je přechodová matice K s prvky

$$P_{xy} = P(X_n = y | X_{n-1} = x), x, y \in (\mathcal{X}).$$

Ve spojitém případě jádro také určuje podmíněnou hustotu $K(x, x')$ přechodu $\mathcal{X}$, $K(x, \dot{)}$, tedy $P(X \in A|x) = \int_A K(x, x')dx'$.

To, že máme přechodové jádro ještě nemusí stačit, proto aby byla posloupnost náhodných veličin Markovovský řetězec musí platit, že podmíněné pravděpodobnostní rozdělení dalšího stavu záleží jen na stavu předchozím.

Definice 2 *Mějme přechodové jádro K, posloupnost $X_0, X_1, ..., X_n, ...$ náhodných veličin je Markovovský řetězec, pokud pro jakékoli t podmíněna distribuce X_t za $x_{t-1}, x_{t-2}, ..., x_0$ je stejná jako distribuce X_t za x_{t-1}, tedy:*

$$P(X_{k+1} \in A|x_0, x_1, x_2, ..., x_k) = P(X_{k+1} \in A|x_k) = \int_A K(x_k, dx).$$

Řetězec je homogenní pokud:

$$P(X_{k+1}|y) = P(X_k|y).$$

Pro využití v MCMC jsou důležité dvě vlastnosti Markovovských řetězců a to neredukovatelnost a aperiodicita. Neredukovatelnost zjednodušeně říká, že Markovovský řetězec se někdy dostane do každého možného stavu nezávisle na tom v jakém stavu jsme začali. Formálně pak

Definice 3 *Nechť ϕ je míra, pak Markovský řetězec (X_n) s přechodovým jádrem $K(x, y)$ je ϕ neredukovatelný pokud, pro všechny $A \in \mathcal{B}(\mathcal{X})$ s $\phi(A) > 0$, existuje n takové, že $K^n(x, A) > 0$ pro všechna $x \in \mathcal{X}$.*
Řetěz je silně ϕ-neredukovatelný pokud $n = 1$ pro všechny měřitelné A.

Aperiodicitou myslíme, že stavy které v řetězci navštěvujeme se periodicky nepakují. V diskrétním případě se perioda stavu $\omega \in \mathcal{X}$ definuje jako:

$$d(\omega) = \gcd m \geq; K^m(\omega, \omega) > 0, \tag{31}$$

kde gcd znamená největší společný dělitel. Říkáme tedy, že neredukovatelný Markovský řetěz je aperiodický pokud má periodu 1. Definice v obecném případě je složitější, proto ji necháváme na vhodnějším zdroji [7].

V MCMC metodách se používají takové řetězce u kterých marginální distribuce X_n nezávisí na n. U takových řetězců, tedy požadujeme, aby existovala pravděpodobnostní distribuce π taková, že $X_{n+1} \sim \pi$ pokud $X_n \sim \pi$. Tato distribuce nám tedy říká s jakou pravděpodobnostní se vyskytuje nějaký stav x v řetězci.

Definice 4 *σ-konečná míra π je invariantní pro přechodové jádro $K(;\dot{)}$ pokud*

$$\pi(B) = \int_{\mathcal{X}} K(x, B)\pi(dx), \forall B \in \mathcal{B}(\mathcal{X}).$$

Invariantní distribuce se také říká stacionární pokud π je pravděpodobnostní míra.

Postačující podmínkou pro existenci stacionární distribuce π, je takzvaná detailed balance podmínka

Definice 5 *Markovovský řetězec s přechodovým jádrem K splňuje detailed balance podmínku pokud existuje funkce π taková, že*

$$K(y, x)\pi(y) = K(x, y)\pi(x), \forall x, y \in \mathcal{X}.$$

Pokud přechodová matice K splňuje podmínky neredukovatelnosti a aperiodicity pak platí, že pro jakýkoli počáteční stav, řetězec konverguje ke stacionární distribuci $\pi(x)$. K zajištění toho, že konkrétní $\pi(x)$ je stacionární distribuce se v MCMC algoritmech používá podmínka 5. Diskuzi obecného případu opět přenecháváme povolanějším [7].

4.3 Metropolis-Hastings

Metropolis-Hastings je nejpopulárnější MCMC algoritmem. Pomocí něj můžeme vzorkovat z libovolné distribuce $\pi(x)$, kterou umíme bodově vyčíslit až na normalizační konstantu. Jeden krok Metropolis-Hastings algoritmu se skládá z navržení nového vzorku x^* z nějaké návrhové distribuce $q(x^*|x)$, tedy návrh nového vzorku x^* je podmíněný předchozím vzorkem x. A poté s pravděpodobností $A(x, x^*) = min(1, \frac{q(x|x^*)\pi(x^*)}{q(x^*|x)\pi(x)})$ příjmáme nový vzorek. Pokud nový vzorek nepříjmeme označíme za nový vzorek původní vzorek x.

Přechodové jádro pro Metropolis-Hastings algoritmus je:

$$K_{MH}(x^{i+1}|x^i) = q(x^{i+1}|x^i)A(x^i, x^{i+1}) = \delta_{x^i}(x^{i+1})r(x^i) \tag{32}$$

kde $r(x^i)$ popisuje pravděpodobnost odmítnutí vzorku:

$$r(x^i) = \int_{\mathcal{X}} q(x^*|x^i)(1 - A(x^i, x^*))dx^*. \tag{33}$$

Přechodové jádro splňuje detailed balance podmínku:

$$\pi(x^i)K_{MH}(x^{i+1}|x^i) = \pi(x^{i+1})K_{MH}(x^i|x^{i+1}) \tag{34}$$

tudíž Markovský řetězec konstruovaný tímto jádrem má $\pi(x)$ jako svoji stacionární distribuci. Pro to, že k ní skutečně dokonverguje, je potřeba vědět, jestli takto zkonstruovaný řetězec je aperiodický a irreducible. Z toho, že přechodové jádro vždy umožňuje nový vzorek odmítnout, následuje to, že řetězec je aperiodický. K tomu aby byl řetězec neredukovatelný, stačí zajistit, že nosič návrhové distribuce je stejný jako nosič $\pi(x)$ [4].

Úspěch algoritmu je hodně závislý na použité návrhové distribuci, může se stát, že pokud použijeme špatnou návrhovou distribuci zůstaneme na jednom místě a nikdy se nepohneme z prvotního vzorku. Naopak výhodou algoritmu je fakt, že distribuci, z které chceme vzorkovat stačí znát bez normalizační konstanty. Tohoto faktu využíváme i v naší aplikaci. Je to z toho důvodu, že normalizační konstanty se objeví v čitateli i jmenovateli při výpočtu pravděpodobnosti přechodu a tudíž se vykrátí.

Samotná implementace algoritmu už je poměrně přímočará, jak je vidět na pseudo-kódu 1. Časová náročnost bude dána počtem vzorků, které chceme získat, ale počtem pozorovaných dat. Pro každý nový vzorek se bude muset napočítat jeho pravděpodobnostní funkce a ta se bude rovnat produktu hustoty pravděpodobnosti aplikované na pozorované data. Tento fakt může vést k dlouhé době vzorkování.

```
1   samples = [first_sample]
2   for i in range(1, n):
3      previous_sample = samples[i-1]
4      new_sample = proposal.propose(previous_sample)
5      u = uniform(0, 1)
6      up = proposal.probablity(new_sample, previous_sample)stationary.
          probability(new_sample)
7      down = proposal.probablity(previous_sample, new_sample)stationary.
          probability(previous_sample)
8      A = up/down
9       if u < A:
10        samples.append(new_sample)
```

```
11          else:
12              samples.append(previous_sample)
```

Výpis 1: Metropoli-Hastings

5 RJMCMC

Nyní se konečně dostáváme k algoritmu, který nám umožní vyřešit náš model. RJMCMC (Reversible Jump Markov Chain Monte Carlo)[5] jako takový nám umožní procházet stavy ve formě $(k, \theta_k) = (k, \theta_{k_1}, ..., \theta_{k_{n_k}})$. Stavový prostor pro takovou simulaci je $\mathcal{X} = \bigcup_{k \in K}(k \times \mathcal{X}_k)$, kde pro každé k je $X_k \subset R^{n_k}$ a K je množina indexů všech přípustných modelů. V našem případě po částech lineární regrese je K množina jdoucí od nuly do nekonečna a každé číslo označuje počet zlomů, takže pro model 1 stav vypadá takto $(1, \theta_1) = (1, \sigma^2, s_1, h_1, s_2, h_2, s_3, h_3)$, kde $\forall s_1, s_2, s_3, h_1, h_2, h_3 \in \mathcal{R}$ a $\sigma^2 > 0$.

Půjde nám, podobně jako v předchozí kapitole, o to, sestavit markovovský řetězec tak, aby jeho stacionární distribuce byla $\pi(x)$, kde $x = (k, \theta_k)$. V konstrukci tohoto řetězce bude třeba navrhovat kroky, které umožní změnu dimenze a taky kroky, které prozkoumají prostor v současném modelu. Pro návrh kroků druhého typu se používají klasické MCMC algoritmy, v našem případě budeme používat algoritmus Metropolis-Hastings a protože jsme se tímto už zabývali v předchozí kapitole nebudeme se tímto již zabývat. Zůstávají nám tedy mezi modelové kroky.

Různé modely ale mohou mít různou dimenzi a toto přináší s sebou problém s porovnáváním. Jak se dá porovnat pravděpodobnost kruhu a koule? Toto se řeší něčím čemu se říká Reversible Jump. Reversible jump se skládá jak z dopředného kroku kroku, která vezme současný stav $x = (k, \theta_k)$ do stavu nového $x' = (k', \theta'_k)$, tak ze zpětného kroku, který provede opačný skok. Tyto přeskoky budeme indexovat v počitatelné množině $\mathcal{M}$ a pro každý přeskok $m \in \mathcal{M}$ máme pravděpodobnost, že se o něj pokusíme v současném stavu x a tu značíme $j_m(x)$. Z každého modelu k, bude obvykle jen několik skoků o které se můžeme pokusit, většinou to budou takové skoky, které budou skákat do nejbližších možných dimenzí. Takže třeba v našem případě povolíme skok jen do modelu, který ma buď o jeden bod více nebo méně. Pro skok vpřed vygenerujeme r_m náhodných čísel ze známé sdružené distribuce g_m a nový stav $\theta'_{k'}$ je konstruován takto $(\theta'_{k'}, u') = h_m(\theta_k, u)$. Tady u' je r'_m náhodných čísel ze sdružené distribuce g'_m, které jsou potřeba pro zpětný krok z $\theta'_{k'}$ do θ_k za použití inverzní funkce h'_m. Důležité pro ně je, aby platilo $n + r = n' + r'$, tedy aby byly diferencovatelné.

V našem případě může jeden skok sloužit například k určení kde se objeví x-ová a y-ová souřadnice bodu, který přibude v modelu s vyšší dimenzí. Takže například:

$$h(\sigma^2, s_1, h_1, s_2, h_2, u_1, u_2) = (\sigma^2, s_1, h_1, u_1(s_2 - s_1), u_2(h_2 - h_1)), \tag{35}$$

$$h'(\sigma^2, s_1, h_1, s_2, h_2, s_3, h_3) = (\sigma^2, s_1, h_1, s_3, h_3, \frac{s_2}{s_3 - s_1}, \frac{h_2}{h_3 - h_1}), \tag{36}$$

$$u_1 \sim U(0, 1), \tag{37}$$

$$u_2 \sim N(0.5, 3). \tag{38}$$

Na našem příkladě vidíme, že podmínka rovnajících se dimenzí je zachována a taky, že u inverzní transformace nemáme žádnou náhodnou veličinu a obecně ani není potřeba **??**.

Veškeré zvláštnosti týkající se RJMCMC jsme si již ukázali jediné co zbývá, je ukázat samotnou přechodovou pravděpodobnost:

$$A_m(x, x') = \frac{\pi(x')j_m(x')g_m'(u')}{\pi(x)j_m(x)g_m(u)} \left| \frac{\partial(\theta_k', u')}{(\theta_k, u)} \right|. \tag{39}$$

Je třeba poznamenat, že Jacobián vstupuje do výrazu jen proto, že návrhová distribuce je určena nepřímo a ne kvůli toho, že se mění dimenze [5].

5.1 Algoritmus

```
1  def trans_step(x, m):
2  k, theta = x
3  u = g_m()
4  new_k, new_theta, new_u = h_m(theta, u)
5  new_x = (new_k, new_theta)
6  up = pi(new_x)j_m(new_x)new_g_m(new_u)
7  down = pi(x)j_m(x)g_m(u)
8  A = det(h_jacobian(new_theta, new_u))*up/down
9  if A < uniform(0, 1):
10     return new_x
11 else:
12    return x
```

Výpis 2: Algoritmus pro mezi dimenzionální skok

Hlavní rozdíl oproti obyčejnému Metropolis Hastingsovi je ten, že musíme přecházet mezi dimenzemi, proto se touto části algoritmu budeme zabývat jako první. h_m je deterministická funkce parametrů theta a náhodného vektoru u. Jak je tato funkce definována záleží na konkrétním modelu. Výsledkem této funkce je trojice $theta'$, u', k', kde $theta'$ je vzorek odpovídající modelu k'. u' je potom náhodný vektor, který by byl potřeba pro zpětnou transformaci, obvykle se stane, že buď u nebo u' bude prázdný vektor. Vytvoření x' je už spíš otázka stylu a implementace stacionární distribuce π. Výstupem ze samotné funkce, ale musí být dvojice (k, θ), tak aby se, potom co je vzorkování ukončeno, dalo zjistit, které vzorky, patří ke kterému modelu.

Z pseudo-kódu 3 se může zdát, že algoritmus je implementačně podobně složitý jako Metropolis-Hastings, není tomu, ale tak. Fakt, že v každém modelu budou možné jiné skoky, přináší problém s tím, jak mezi nimi vybrat a jak vybrat jen ze skoků určených pro konkrétní model. Dalším problémem je, jak napočítat determinanty Jakobiánů transformačních funkcí. Ručně je toto velmi pracné a chybám náchylné, proto doporučujeme použít nějakou knihovnu, která umožní Jakobiány spočítat symbolicky. I zde samozřejmě platí poznámky týkající se časové náročnosti algoritmu Metropolis-Hastings.

```
def rjmcmc(first_sample, n):
samples = [first_sample]
for i in range(1, n):
   previous_sample = samples[i-1]
   if do_trans_step:
      choose move from moves
      new_sample = trans_step(previous_sample, move)
   else:
      new_sample = mcmc_step(previous_sample)
   samples.append(new_sample)
```

Výpis 3: Celkový RJMCMC algoritmus

Pseudokód 3 popisuje jak funguje celý algoritmus RJMCMC. V tom, kdy se pokusit o přechod mezi dimenzemi, máme značnou volnost, jedna možnost je určit pokus o takovýto krok deterministicky, takže například každou desátou iteraci se pokusíme o přeskok. Při výběru o jaký přeskok se pokusíme, postupujeme obdobně. Při výběru konkrétního kroku je asi implementačně jednodušší vybírat typ kroku náhodně podle předem zadaných pravděpodobností. A to z toho důvodu, že možností jak změnit dimenzi může být značné množství, ale ne každý krok je přístupný z každé dimenze. U kroků, které nemají být přístupny z dané dimenze, můžeme nastavit nulovou pravděpodobnost a tím zajistíme, že se o ně nikdy nepokusíme. Pokud se nepokoušíme o přechod mezi dimenzemi, použijeme treba MH pro získání dalšího vzorku v rámci současného modelu.

5.2 Příklad

Na simulovaných datech si nyní ukážeme, jak algoritmus funguje i s jeho nastavením. Algoritmus RJMCMC jsme implementovali v programovacím jazyce Python[2] za využití knihoven NumPy[1] a SciPy**??**. Tyto knihovny jsme využili především kvůli toho, že je v nich dobrá dostupnost různých pravděpodobnostních rozdělení, které se dají využít jak k výpočtu pravděpodobnostních hustot, tak ke generování vzorku z daných distribucí. Dále jsme použili knihovnu Theano[3] a to k implementaci transformačních funkcí a především k symbolickému výpočtu jejich Jakobiánů.

Model bude vždy stejný a to sice model zadefinovaný zde 27. Omezíme se na model s maximálně dvěma zlomy. Přechod mezi modelem beze zlomu a modelem s jedním zlomem a zpátky

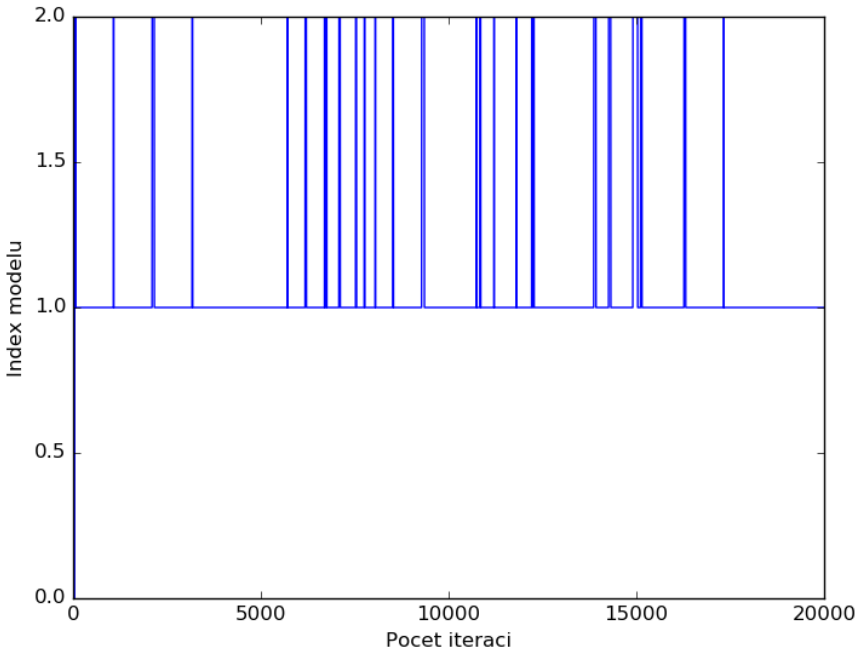
vypadá následovně:

$$h_1(\sigma^2, s_1, h_1, s_2, h_2, u, n) = (\sigma^2, s_1, h_1, s_1 + (s_2 - s_1)u, h_1 + (h_2 - h_1)u_1 + n, s_2, h_2),$$
$$kde\ u \sim U(0,1),\ n \sim N(0,3)$$
$$h_1^{\cdot}(\sigma^2, s_1, h_1, s_2, h_2, s_3, h_3) = (\sigma^2, s_1, h_1, s_3, h_3, u, h_2 - h_1 - (h_3 - h_1)u),$$
$$kde\ u = \frac{s_2 - s_1}{s_3 - s_1}.$$

Náhodná veličina u slouží k určení, kde mezi počátečním a koncovým bodem by se měl nacházet nový bod. Náhodná veličina n určuje to, o kolik výš nebo níž by měl nový bod být oproti původní přímce. Jak je vidět v kroku zpět se nepoužívají žádné náhodné veličiny, ale dopočítavají se jaké by musely být hodnoty u a n, kdybychom chtěli přejít z jednoduššího modelu do složitějšího. Tyto hodnoty je důležité vědet, protože i na nich závisí jaká bude přechodová pravděpodobnost, jak je vidět v 39. Pro přechod mezi modely s jedním zlomem a dvěma a zpět používáme podobnou transformaci:

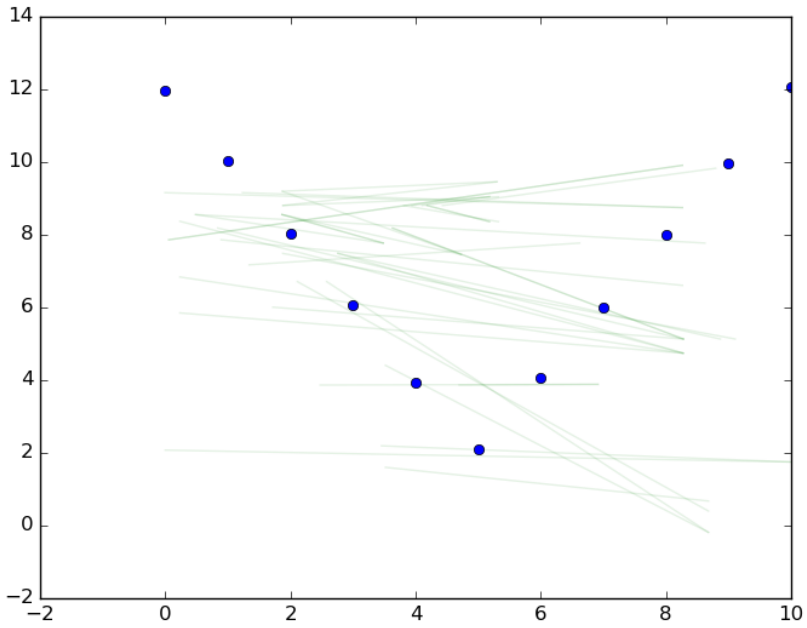
$$h_1(\sigma^2, s_1, h_1, s_2, h_2, s_3, h_3, u, n) = (\sigma^2, s_1, h_1, s_2, h_2, s_2 + (s_3 - s_2)u, h_2 + (h_3 - h_2)u + n, s_3, h_3),$$
$$kde\ u \sim U(0,1),\ n \sim N(0,3)$$
$$h_1^{\cdot}(\sigma^2, s_1, h_1, s_2, h_2, s_3, h_3, s_4, h_4) = (\sigma^2, s_1, h_1, s_2, h_2, s_4, h_4, u, h_3 - h_2 - (h_4 - h_2)u),$$
$$kde\ u = \frac{s_3 - s_2}{s_4 - s_2}.$$

Jako návrhovou distribuci v rámci přechodů uvnitř modelu beze zlomů jsme zvolili $q(x^*|x) \sim \mathcal{N}(x, 5I)$, kde I je jednotková matice 5×5, s jednou modifikací a to, že pro σ^2 se přijímají jen vzorky větší než nula. Podobné návrhové distribuce se používají v modelech s vyšší dimenzí. Simulované data pro tento příklad jsou $x = (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10)$, $y = (12 + e_1, 10 + e_2, 8 + e_3, 6 + e_4, 4 + e_5, 2 + e_6, 4 + e_7, 6 + e_8, 8 + e_9, 10 + e_1 0, 12 + e_1 1)$, kde $e_i \sim \mathcal{N}(0, 0.1)$. Algoritmus jsme nechali běžet o 20000 iterací. A prvotní vzorek je náhodně vygenerovaný vzorek začínající v modelu beze zlomu.

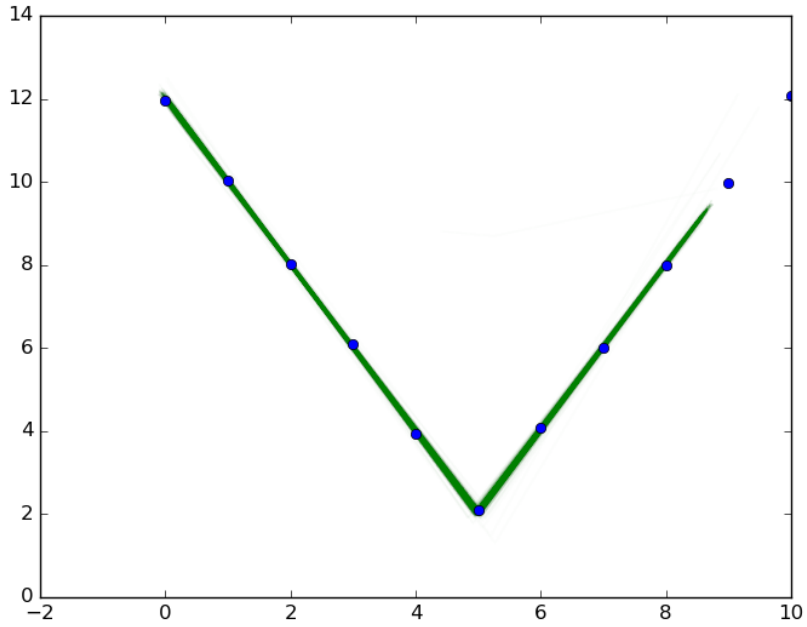


Obrázek 1: Vykreslení přeskoku mezi dimenzemi

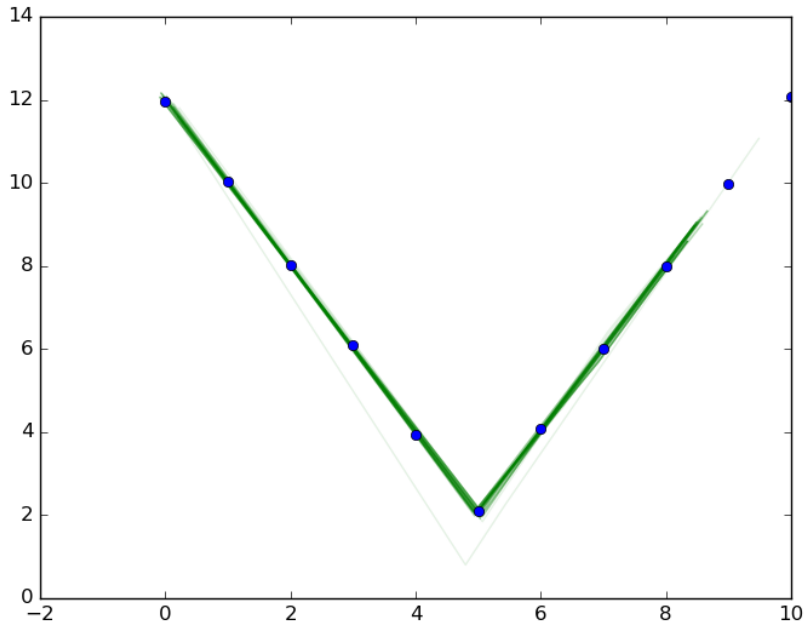
Z vykreslení přeskoků mezi dimenzemi 1 je patrné, že algoritmus zůstává v modelu beze zlomu (index 0), prakticky jen nezbytně dlouho dobu než dostane možnost přeskočit do jiné dimenze a poté se už do modelu beze zlomu nikdy nevrací. Poté sice přeskakuje mezi modelem s jedním a dvěma zlomy, ale většinu času stráví v modelu s jedním zlomem. Celkově algoritmus získal 39 vzorků v modelu beze zlomu, 19233 v modelu s jedním zlomem a 728 vzorků v modelu s dvěma zlomy.



Obrázek 2: Aproximace distribuce modelu beze zlomu



Obrázek 3: Aproximace distribuce modelu s jedním zlomem

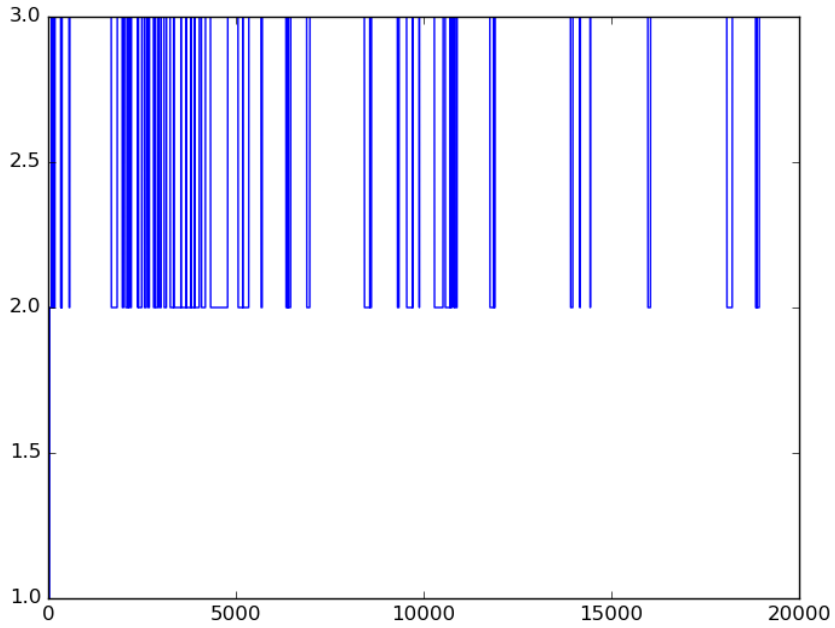


Obrázek 4: Aproximace distribuce modelu s dvěma zlomy

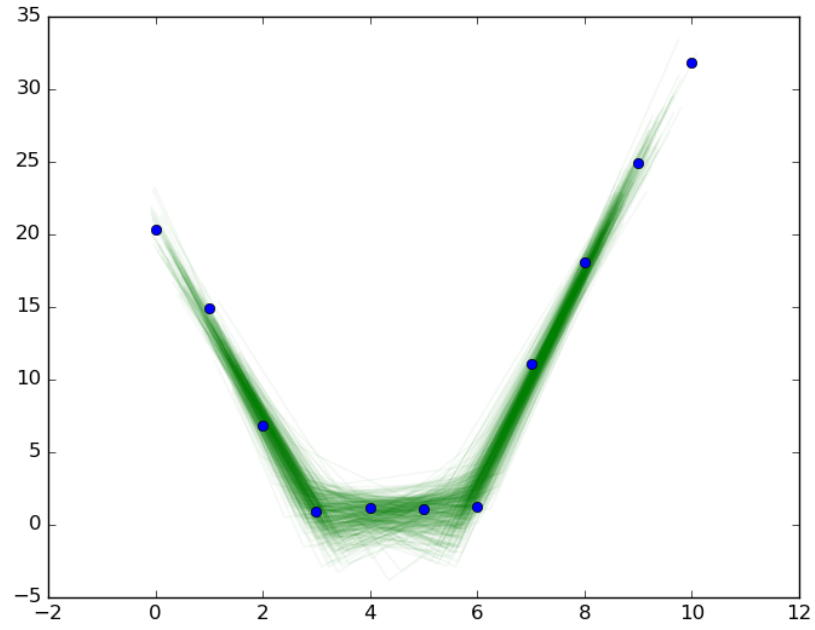
Na obrázcích 2, 3, 4 můžeme vidět aproximace distribucí pro dané modely, pomocí vzorků z jednotlivých modelů. Zajímavé je si všimnou, faktu, že i když distribuce pro model s jedním respektive dvěma zlomy vypadají podobně, tak algoritmus strávil mnohem víc času v modelu s jedním zlomem. Je to právě dáno bayesovským přístupem, který zvýhodňuje jednodušší modely se stejnou vysvětlovací schopností.

Ukážeme si ještě jeden příklad a to se stejnými $x = (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10)$ jako minule, ale $y = (20 + e_1, 15 + e_2, 7 + e_3, 1 + e_4, 1 + e_5, 1 + e_6, 1 + e_7, 11e_8, 18 + e_9, 25 + e_10, 32 + e_11)$ a $e_i \sim \mathcal{N}(0, 0.1)$ respektive $e_i \sim \mathcal{N}(0, 2)$.

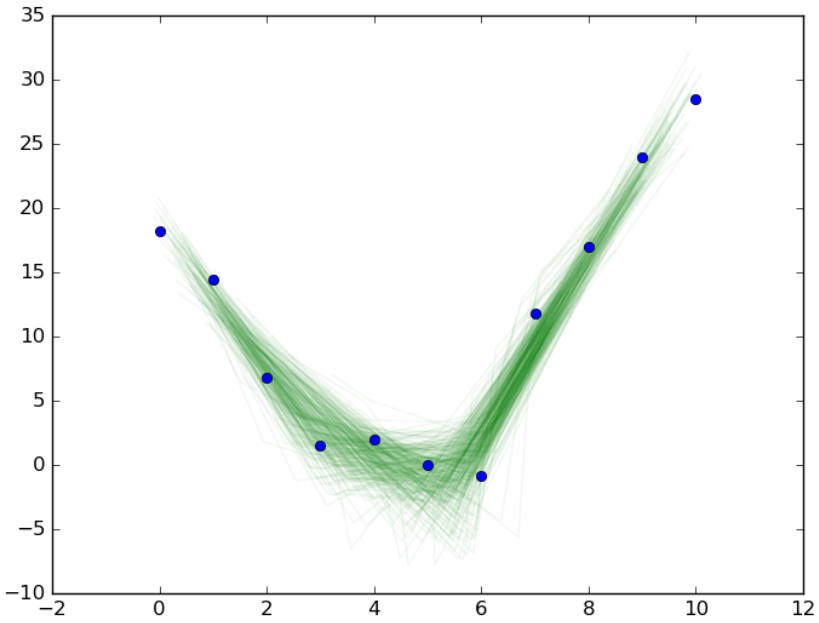
V prvním případě algoritmus prakticky okamžitě přechází do modelu s dvěma zlomy a už z něj nevychází, počet vzorků v modelu 0 je 65, modelu 1 78, modelu 2 19857. V případě s vyšší dimenzí je rozdíl a to, že o hodně častěji algoritmus přechází do modelu 1. Počet vzorků v modelu 0 je 38, modelu 1 4039, modelu 2 15923. Na obrázku 5 je vidět jak algoritmus přechazí mezi jednotlivými modely.



Obrázek 5: Vykreslení přeskoku mezi dimenzemi příklad 2, $\sigma^2 = 2$



Obrázek 6: Aproximace distribuce modelu s dvěma zlomy, $\sigma^2 = 0.1$



Obrázek 7: Aproximace distribuce modelu s dvěma zlomy, $\sigma^2 = 2$

Na obrázcích 6, 7 je vidět distribuce s menším respektive větším rozptylem. Na distribuci s menším rozptylem je vidět, že mnohem více koncentrována kolem skutečných bodů zlomu, což jsou body 3 a 6. A naopak distribuce s větším rozptylem není tak moc koncentrovaná, jak se dalo čekat.

6 Závěr

V práci jsme stručně popsali problémy, které se pojí s řešením po částech lineární regrese. Začli jsme úvodem do Bayesovské statistiky. V jejímž rámci jsme poté zavedli model pro po částech lineární regresi. Následně jsme si ukázali jak fungují MCMC algoritmy, které ale samy o sobě nejsou dostačující pro dosažení závěrů o našem modelu, proto jsme přešli k algoritmu RJMCMC. Ten dokáže aproximovat aposteriorní rozdělení u modelů, kde jeden z parametrů je počet ostatních parametrů. Stručně jsme popsali to jak algoritmus RJMCMC, řeší problém s přechodem mezi dimenzemi a ukázali jsme jak by takové přechody mohly vypadat.

Značná část naší práce spočívá v samotné implementaci algoritmu RJMCMC. Ten jsme naprogramovali v jazyce Python za pomocí knihoven NumPy a SciPy, které se používají obecně k vědeckým výpočtům a v našem případě konkrétně jsme je použili pro simulaci dat z různých pravděpodobnostních rozdělení. Také jsme použili knihovnu Theano, která slouží k symbolickým výpočtům derivací a v našem případě Jakobiánu. Na závěr naší práce jsme předvedli funkčnost naší implementace algoritmu na simulovaných datech a popsali výsledky.

Protože jsme přesvědčení, že naše implementace algoritmu RJMCMC je dosti obecná, přirozeným pokračováním naší práce by bylo vyzkoušení daného algoritmu na jiných problémech, jako například rozpoznávaní obrazů nebo pro směsi normálních rozdělení. Dále by si tato práce zasloužila rozvést teoretickou stránku fungování algoritmu. Protože většina reálných problému nezávisí jen na jedné veličině, bylo by dalším zajímavým rozšířením pokusit se implementovat algoritmus RJMCMC pro více dimenzionální lineární regresi. Toto by už by mohlo být značně náročné z hlediska návrhu návrhových distribucí.

Literatura

[1] Numpy. http://www.numpy.org/.

[2] Python. https://www.python.org/.

[3] Theano. http://deeplearning.net/software/theano/.

[4] Christophe Andrieu, Nando De Freitas, Arnaud Doucet, and Michael I Jordan. An introduction to mcmc for machine learning. *Machine learning*, 50(1-2):5–43, 2003.

[5] Peter J Green and David I Hastie. Reversible jump mcmc. *Genetics*, 155(3):1391–1403, 2009.

[6] Christian Robert. *The Bayesian choice: from decision-theoretic foundations to computational implementation*. Springer Science & Business Media, 2007.

[7] Christian P Robert. *Monte carlo methods*. Wiley Online Library, 2004.

A Zdrojové kódy

```
class Rjmcmc:
'''
trida ktera zastitute cely reversible jump markov chain
monte carlo algoritmus
'''

   def __init__(self, moves, mcmcs, stationary):
      self.moves = moves
      self.mcmcs = mcmcs
      self.stationary = stationary
      self.trans_steps = 0
      self.norm_steps = 0

   def step(self, previous_sample):
      k, theta = previous_sample
      u = uniform()
      moves = [m for m in self.moves if m.can_move(k) > 0]
      trans_probability = 0
      for m in moves:
         trans_probability += m.probability_of_this_move(previous_sample)
      if u < trans_probability:
         self.trans_steps += 1
         uu = uniform(0, trans_probability)
         M = None
         prob = 0
         for m in moves:
            if prob < uu < prob + m.probability_of_this_move(previous_sample):
               M = m
               break
            prob += m.probability_of_this_move(previous_sample)

         up, down, a, new = self.trans_step(M, previous_sample)
         return new
      else:
         self.norm_steps += 1
         if k*2 + 3 is not len(theta):
            print(k)
```

```
38            print(previous_sample)
39            raise AssertionError
40         return (k, self.mcmcs[k].step(theta))
41
42   def trans_step(self, move, previous_sample):
43      (k, theta) = previous_sample
44
45      new_sample, u, newu, det_jacobian = move.transform(previous_sample)
46
47      up = np.prod([self.stationary(new_sample),
48      move.probability_of_this_move(new_sample),
49      move.probability_of_help_rvs(new_sample, newu)])
50      down = np.prod([self.stationary(previous_sample),
51      move.probability_of_this_move(previous_sample),
52      move.probability_of_help_rvs(previous_sample, u)])
53
54      a = det_jacobian*up/down
55      u = uniform()
56
57      if u < a:
58         return (up, down, a, new_sample)
59      else:
60         return (up, down, a, previous_sample)
```

Výpis 4: RJMCMC

```
class Blr2:
    '''
    Trida, ktera vytvori stacionarni distribuci pro regresi s n zlomy.
    2 protoze pouzivam jinou parametrizaci.
    sigma = x[0] - rozptyl no :D
    s0 = x[1] - xova souradnice prvniho zlomu
    h0 = x[2] - yova souradnice prvnhio zlomu
    ...
    sn = x[n-2] - xova souradnice nteho zlomu
    hn = x[n-1] - yova souradnice nhteo zlomu
    '''

    def __init__(self, xs, ys, n_breaks):
        '''
        @param xs - xove souradnice dat
        @param ys - yove souradnice dat
        @param n_breaks - pocet zlomu
        '''
        if len(xs) is not len(ys):
            raise RuntimeError("Not matchin dimension")
        self.xs = xs
        self.ys = ys
        self.max_x = max(xs)
        self.min_x = min(xs)
        self.n = 2*n_breaks + 5
        self.n_samples = len(xs)
        self.h_prior = normal(np.zeros(int((self.n-1)/2)),
                              100*np.eye(int((self.n-1)/2)))
        self.sigma_prior = normal(0, 3)
        self.n_breaks = n_breaks

    def prior_s(self, theta):
        '''
        Apriorni rozdeleni na thetaovych souradnicich. Tedy melo by platit
        ss < s1 < s2 < ... < sn < sf. Je to tak nastaveno z toho duvodu,
        aby bylo dodrzeni poradi
        '''
        # taky si nejsem jisty jestli prochazim vsechny
```

```
39          x_coordinates = [theta[i] for i in range(1, self.n, 2)]
40          previous = x_coordinates[0]
41          for i in range(1, len(x_coordinates)):
42              if previous > x_coordinates[i]:
43                  return 0
44              previous = x_coordinates[i]
45
46          if x_coordinates[0] < self.min_x - 0.1:
47              return 0
48          if x_coordinates[len(x_coordinates) - 1] > self.max_x + 0.1:
49              return 0
50          return 1
51
52      def prior_h(self, theta):
53          '''
54          Apriorni rozdeleni na yovych souradnicich. Je teda co nejvic
55          neinformativni, tedy pro vsechny h plati, ze h ~ N(0, 100)
56          '''
57          # tady se trochu bojim ze neprojdu vsechny
58          # jestli se dostanu za hranici tak se to rychle odhali :D
59          y_coordinates = [theta[i] for i in range(2, self.n, 2)]
60          return self.h_prior.pdf(y_coordinates)
61
62      def prior_sigma(self, theta):
63          '''
64          Apriorni rozdeleni na rozptylu. Zas jen nake neinformativni a s
65          nulovou pravdepodobnosti na sigmach mensi nez 0
66          '''
67          if theta[0] > 0:
68              return self.sigma_prior.pdf(theta[0])
69          return 0
70
71      def likelihood(self, theta):
72          '''
73          Spocita likelihood hustotu pro dany vzorek
74          '''
75          assert len(theta) == self.n
76
77          suma = 0
```

```
78          for i, xi in enumerate(self.xs):
79              yi = self.ys[i]
80              for j in range(1, self.n-2, 2):
81                  break1 = (theta[j], theta[j+1])
82                  break2 = (theta[j+2], theta[j+3])
83
84                  if break1[0] <= xi < break2[0]:
85                      suma += self.prob_sum(xi, yi, break1, break2)
86
87              # tohle je pro pripad ze xi je pred nebo za body urcujicimi primku
88              # stava se to :D
89              if xi < theta[1]:
90                  break1 = (theta[1], theta[2])
91                  break2 = (theta[3], theta[4])
92                  suma += self.prob_sum(xi, yi, break1, break2)
93
94              if xi > theta[self.n - 2]:
95                  break1 = (theta[self.n-4], theta[self.n-3])
96                  break2 = (theta[self.n-2], theta[self.n-1])
97                  suma += self.prob_sum(xi, yi, break1, break2)
98
99          try:
100             exp = np.exp(-suma/(2*theta[0]))
101             bs = theta[0]**(-len(self.xs)/2)
102             return bs * exp
103         except FloatingPointError:
104             print()
105             print('theta0 ' + str(theta[0]))
106             print('suma ' + str(suma))
107             return 0
108
109     def prob_sum(self, x, y, break1, break2):
110         '''
111         pomocna funkce, co mi spocita jeden vyraz v exponenciale, jakoze v
112         Normalnim rozdeleni hore
113         '''
114
115         # a = (break2[1] - break1[1])/(break2[0]-break1[0])
116         # b = break2[1] - break2[0]*a
```

```
117         x1, y1 = break1
118         x2, y2 = break2
119         est = (x - x1)*(y2 - y1)/(x2 - x1) + y1
120         return (y - est)**2
121 
122     def pdf(self, theta):
123         if len(theta) is not self.n:
124             print(theta)
125             raise Exception("Co to kurva")
126 
127         prior_probs = np.prod([self.prior_h(theta),
128                                self.prior_s(theta),
129                                self.prior_sigma(theta)])
130 
131         # netkere vzorky budou mit nulovou pravdepodobnost
132         # uz kvuli apriornimu rozdeleni, proto to checknu
133         # at se nemusi pocitat likelihood ten v zavislosti
134         # na datech muze byt dost narocny spocitat
135         if prior_probs == 0:
136             return 0
137         return np.prod([prior_probs,
138                         self.likelihood(theta)])
139 
140     def generate_first_sample(self):
141         '''
142         vygeneruje nejaky vzorek, ktery nema pravdepodobnost nula
143         '''
144         minimum = min(self.xs)
145         maximum = max(self.xs)
146         first_sample = np.zeros(self.n)
147 
148         first_sample[0] = 1
149 
150         for i, x in enumerate(np.linspace(minimum, maximum, (self.n-1)/2)):
151             first_sample[2*i + 1] = x
152             first_sample[2*i + 2] = np.random.normal(0, 3)
153 
154         if not self.pdf(first_sample) > 0:
155             print("First sample: " + str(first_sample) +
```

```
156                 " has zero probability")
157             print("prior h " + str(self.prior_h(first_sample)))
158             print("prior s " + str(self.prior_s(first_sample)))
159             print("prior sigma " + str(self.prior_sigma(first_sample)))
160             print("likelihood " + str(self.likelihood(first_sample)))
161             return self.generate_first_sample()
162 
163         return first_sample
```

Výpis 5: implementace aposteriorního rozdělení pro po částech lineární regresi

```python
class Mcmc:
    def __init__(self, proposalDistribution, stationaryDistribution):
        self.proposal = proposalDistribution
        self.stationary = stationaryDistribution

    def step(self, previous_sample):
        '''
        Vnitrek mcmc algoritmu, prakticky to co se deje v jedne
        iteraci tady te verze hastingse
        '''
        proposal_sample = self.proposal.rvs(previous_sample)
        assert proposal_sample is not None
        local_previous_sample = copy.copy(previous_sample)
        final_sample = copy.copy(previous_sample)

        for j, x in enumerate(proposal_sample):
            local_previous_sample[j] = x
            down = np.prod([
                self.stationary.pdf(previous_sample),
                self.proposal.pdf(local_previous_sample, previous_sample)])
            up = np.prod([
                self.stationary.pdf(local_previous_sample),
                self.proposal.pdf(previous_sample, local_previous_sample)])

            u = np.random.uniform()

            if u < up/down:
                final_sample[j] = x
            else:
                local_previous_sample[j] = previous_sample[j]

        return final_sample

    def sample(self, n, first_sample):
        dimension = len(first_sample)
        samples = np.empty((n, dimension))
        samples[0] = first_sample
        # mozna generator?
```

```
39          for i in range(1, n):
40              samples[i] = self.step(samples[i-1])
41              # progress bar
42              sys.stdout.write("\r\t%.0f%% Done" % (100*i/n))
43              sys.stdout.flush()
44          # progress konec
45          sys.stdout.write("\r\t100% Done\n")
46          sys.stdout.flush()
47          return samples
```

Výpis 6: implementace mcmc algoritmu

```
class Move:
    '''
    predstavuje prechod z k do k' a zpet
    '''

    def __init__(self,
                 k1,
                 k2,
                 k1_to_k2,
                 k2_to_k1,
                 transform1to2,
                 transform2to1,
                 jacobian1to2,
                 jacobian2to1,
                 ugenerator1to2,
                 ugenerator2to1,
                 usize1,
                 usize2):
        '''
        k1 - prvni stav
        k2 - druhy stav
        k1_to_k2 - pravdepodobnost prechodu z k1 do k2
        k2_to_k1 - pravdepodobnost prechodu z k2 do k1
        transform1to2 - pretransformuje vzorek na vzorek s jinou dimenzi
                        vraci novy vzorek a mozne nahodne cisla, ktere
                        by byly vegenerovane pro zpetnou transformaci
        transform2to1 - ---||---
        ugenerator1to2 - generator pomocnych nahodnych velicin pro prechod
                         z k1 do k2
        ugenerator2to1 - ---||---
        usize1 - delka vektoru nahodnych velicin ktery se vygeneruje pro
            prechod ze
                 stavu 1 do stavu 2
        usize2 - ---||---
        '''
        self.k1 = k1
        self.k2 = k2
        self.k1_to_k2 = k1_to_k2
```

```
38          self.k2_to_k1 = k2_to_k1
39          self.transform1to2 = transform1to2
40          self.transform2to1 = transform2to1
41          self.jacobian1to2 = jacobian1to2
42          self.jacobian2to1 = jacobian2to1
43          self.ugenerator1to2 = ugenerator1to2
44          self.ugenerator2to1 = ugenerator2to1
45          self.usize1 = usize1
46          self.usize2 = usize2
47
48      def can_move(self, k):
49          '''
50          urci jestli muzu pouzit tento prechod z daneho stavu
51          '''
52          if k == self.k1:
53              return self.k1_to_k2
54          elif k == self.k2:
55              return self.k2_to_k1
56          else:
57              return 0
58
59      def probability_of_this_move(self, x):
60          '''
61          pravdepodobnost pouziti tohole typu pohybu z daneho
62          stavu. zatim pouzivame jen dany stav k, ale v algoritmu
63          je mozne i pouzit pro vypocet pravdepodobnosti soucasny
64          vzorek. takze ted ty prechodove pravdepodobnosti jsou jenom
65          konstanty, ale obecne by to mohla byt i funkce
66          '''
67          k, theta = x
68          if self.k1 == k:
69              return self.k1_to_k2
70          elif self.k2 == k:
71              return self.k2_to_k1
72          else:
73              raise RuntimeError("This should never happen")
74
75      def _transform(self, k, newk, theta, generator, transform, newu_size):
76          if generator is not None:
```

```
77              u = generator.rvs()
78              ext_theta = np.append(theta, u)
79          else:
80              u = None
81              ext_theta = theta
82          newtheta = transform(ext_theta)
83
84          if newu_size is not 0:
85              newu = newtheta[-newu_size:]
86              newtheta = newtheta[:-newu_size]
87          else:
88              newu = None
89          newx = (newk, newtheta)
90          det_jacobian = self.get_jacobian((k, ext_theta))
91          return (newx, u, newu, det_jacobian)
92
93      def transform(self, x):
94          # tady by se rovnou mohl vracet i ten jacobian at se s tim neseru venku
95          k, theta = x
96          if self.k1 == k:
97              return self._transform(k,
98                                     self.k2,
99                                     theta,
100                                    self.ugenerator1to2,
101                                    self.transform1to2,
102                                    self.usize2)
103         elif self.k2 == k:
104             return self._transform(k,
105                                    self.k1,
106                                    theta,
107                                    self.ugenerator2to1,
108                                    self.transform2to1,
109                                    self.usize1)
110         else:
111             raise RuntimeError("This shoould never happen")
112
113     def probability_of_help_rvs(self, x, u):
114         k, theta = x
115         if self.k1 == k:
```

```
116             if self.ugenerator1to2 is None:
117                 return 1
118             return self.ugenerator1to2.pdf(u)
119         elif self.k2 == k:
120             if self.ugenerator2to1 is None:
121                 return 1
122             return self.ugenerator2to1.pdf(u)
123         else:
124             raise RuntimeError("This should never happen")
125 
126     def get_jacobian(self, x):
127         k, theta = x
128         if self.k1 == k:
129             mat = self.jacobian1to2(theta)
130             return np.linalg.det(mat)
131         elif self.k2 == k:
132             mat = self.jacobian2to1(theta)
133             return np.linalg.det(mat)
134         else:
135             raise RuntimeError("This should never happen")
```

Výpis 7: Implementace přeskoků mezi dimenzemi