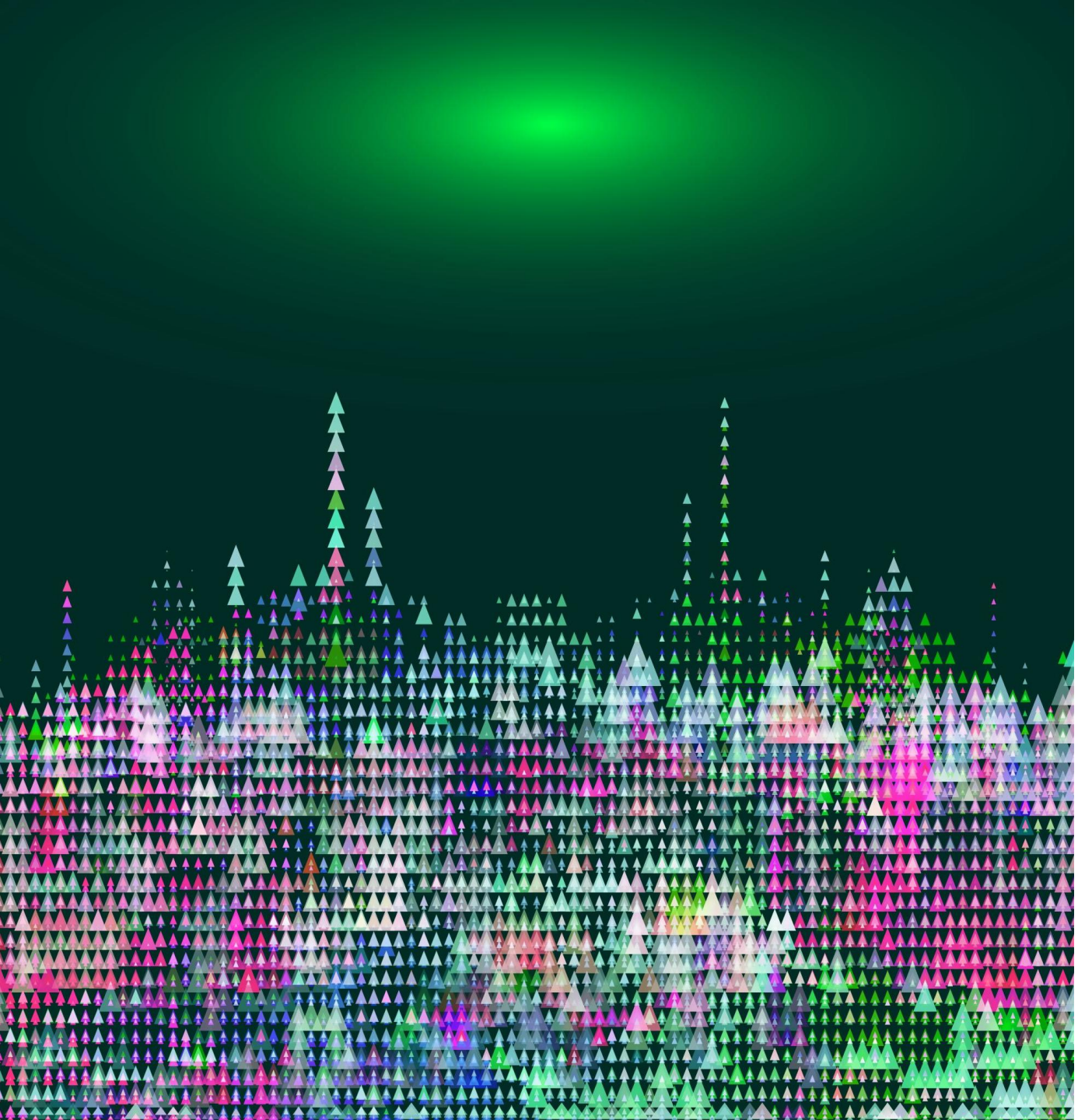

Parametri si Hyperparametri în dezvoltarea aplicațiilor

Conf. dr. Cristian Kevorchian
Facultatea de Matematică și Informatică
Universitatea București

Indicele de mentenanță

Indicele de mentenanță este un indicator compozit care oferă o estimare numerică a cât de ușor poate fi înțeles, modificat și întreținut un fragment de cod. Este un indicator recunoscut global, folosit în multe instrumente de analiză statică (ex: SonarQube, NDepend, CodeClimate).
Este o valoare numerică între 0 și 100, calculată pe baza mai multor metrici de calitate a codului, formula clasică (conform Microsoft) este:

$$\mathbf{IM=max(0,\ (171-5.2\cdot ln(LOC)-0.23\cdot CC-16.2\cdot ln(Volumul\ Halstead))\cdot 100\ \cdot\ 171^{-1})}$$

unde:

- LOC = Linii de Cod
- CC = Complexitatea ciclomatică
- Volumul Halstead(**Michel Halstead**, 1977) este o **măsură a complexității semantice a codului (bazată pe operatori și operanzi):**

$$V=N\cdot \log_2(n)$$

unde:

$$N = N_1 + N_2$$

N_1 = numărul total de operatori (ex: +, if, return)
N_2 = numărul total de operanzi (ex: variabile, constante)

$$n = n_1 + n_2$$

n_1 = numărul de operatori unici
n_2 = numărul de operanzi unici

Valori Prag pentru Indicele de Mentenanță

Indicator de Mentenanță	Descriere	Valori prag comune
MTBF (Mean Time Between Failures)	Timpul mediu între defecțiuni	> 500 ore (ideal)
MTTR (Mean Time To Repair)	Timpul mediu de remediere	< 2 ore (eficient)
Disponibilitatea echipamentului	Procentul de timp în care echipamentul funcționează	> 95% (optim)
Costul mentenanței	Cheltuieli pentru întreținere	< 10% din bugetul total
Procentul de mentenanță preventivă	Raportul între mentenanța planificată și cea reactivă	> 70% (ideal)

Intervalul Indice Mentenanță	Interpretare	Factori care il influențează
0 - 9	Cod dificil de întreținut	Complexitate mare, cuplaj ridicat, moștenire profundă
10 - 19	Mentenanță moderată	Cod cu structuri medii, dependențe gestionabile
20 - 100	Cod ușor de întreținut	Simplitate, modularitate, principii OOP bine aplicate

Complexitatea ciclomatică

- Complexitatea ciclomatică este o măsură utilizată în ingineria software pentru a determina nivelul de complexitate al unui program sau al unui fragment de cod. **A fost introdusă de Thomas J. McCabe în 1976 și se bazează pe analiza grafului de control aferent fluxului de execuție al unui program**.
- Complexitatea ciclomatică se definește ca fiind numărul de drumuri independente aferente unui program, ceea ce înseamnă, practic, numărul minim de teste necesare pentru a verifica fiecare cale posibilă. Se calculează astfel:

$$V(G)=E-N+2P$$

unde

E este numărul de muchii din graful de control al fluxului,
N este numărul de noduri,
P este numărul de componente conexe (de obicei 1 pentru o funcție unică).

- Indică numărul minim de cazuri de test necesare pentru a obține o acoperire completă a codului.
- Ajută la identificarea unor regiuni din cod care sunt prea complexe și dificil de întreținut.
- Poate fi folosită pentru a detecta punctele unde ar putea apărea erori în cod.
  Semnificația valorilor de prag:
  1. **1 – 10 : Cod simplu și ușor de înțeles.**
  2. **11 - 20: Cod cu complexitate moderată, care ar putea necesita o revizuire atentă**
  3. **21 - 50: Cod complex, cu riscuri crescute de erori și dificultăți în testare și întreținere.**
  4. **>50: Cod foarte dificil de gestionat, refactory recomandat**

Adâncimea inferenței

Măsoară cât de complexă este logica de inferență într-un cod, cu alte cuvinte **cât de adânc trebuie să meargă analiza** pentru a determina comportamentul codului analizat.

	Logică și Inteligență Artificială	**Inginerie Software**
Obiectiv	Generarea de concluzii logice	Evaluarea mentenanței codului
Proces	Inferență bazată pe reguli și modele	Analiză structurală și funcțională
Utilizare	Motoare de inferență, sisteme expert, rețele neuronale	Evaluarea mentenanței, analiza dependențelor, optimizare cod
Impact	Precizie vs. cost computațional	Claritate vs. dificultăți de modificare
Reducerea adâncimii	Poate duce la concluzii incomplete	Îmbunătățește mentenanța și debugging-ul

Valorile de prag:

	Mică (1-3)	**Mediu** (4-7)	**Mare** (8+)
Logică și Inteligență Artificială	Inferență rapidă, dar superficială	Raționament echilibrat	Analiză profundă, dar costisitoare computațional
Inginerie Software	Cod clar, ușor de modificat	Complexitate moderată	Cod greu de întreținut, recomandată refactorizarea

Cuplarea Claselor

Cuplarea claselor este un concept esențial în software design, referindu-ne la **gradul de dependență dintre două sau mai multe clase**. Un sistem bine proiectat trebuie să conserve cuplarea slabă, ceea ce facilitează mentenanța, testarea și reutilizarea codului dezvoltat.

Tip de cuplare	Caracteristici	Impact asupra codului
Cuplare puternică	Clasele depind puternic una de alta, modificările într-o clasă afectează cealaltă	Greu de întreținut și testat, reduce considerabil reutilizarea codului
Cuplare slabă	Clasele au puține dependențe directe, interacționează prin abstractizări sau interfețe	Crește flexibilitatea și testabilitatea codului
Cuplare de conținut	O clasă modifică intern datele alteia	Foarte problematică, trebuie evitată
Cuplare de control	O clasă influențează fluxul de execuție al alteia	Limitată recomandată, poate reduce modularitatea
Cuplare de date	Clasele comunică prin parametri bine definiți	O bună practică, susține modularitatea și extensibilitatea

Valorile de prag în cuplarea claselor

Nivel de cuplare	Descriere	Impact asupra codului
Cuplare foarte slabă (0-3)	Clasele sunt independente sau comunică prin interfețe minimale	Modularitate excelentă, ușor de testat și extins
Cuplare moderată (4-7)	Există unele interdependențe, dar logica rămâne clară și controlabilă	Flexibilitate rezonabilă, necesită atenție la modificări
Cuplare puternică (8+)	Clasele sunt interdependente, modificările într-una afectează cealaltă	Mentenanță dificilă, debugging complex, recomandată refactorizarea

Linii de Cod și Linii de Cod executate

	Linii de cod (LoC)	Linii de cod executate
Definiție	Numărul total de linii scrise în codul sursă	Numărul de linii efectiv executate într-un program
Conținut	Comentarii, declarații, instrucțiuni, cod redundant	Doar instrucțiunile care sunt efectiv rulate la execuție
Impact asupra software-ului	Indicativ al dimensiunii proiectului, nu neapărat al eficienței	Determină performanța și acoperirea execuției codului
Utilizare	Estimarea complexității și efortului de dezvoltare	Analiza eficienței, acoperirea testelor și identificarea codului mort
Exemple	Un program poate avea 5000 LoC, dar nu toate sunt necesare pentru rulare	Doar 2000 de linii pot fi executate efectiv, restul pot fi condiționale sau neutilizate

Pragurile și interpretarea acestora

Indicator	Mic (0 - 1000)	Mediu (1001 - 5000)	Mare (>5000)
LOC (Linii de Cod)	Proiect mic, clar, ușor de întreținut	Aplicație medie, necesită organizare eficientă	Proiect mare, posibil greu de gestionat
LOC Executabile	Cod eficient, fără redundanță	Proces optimizat, dar poate conține cod inutilizat	Posibilă supraîncărcare, risc de performanță scăzută

Interpretare

•**LOC mic** → Cod clar, mentenanță ușoară, dar poate fi prea simplist pentru aplicații complexe.
•**LOC mare** → Poate semnala un proiect robust, dar și un cod greu de gestionat, necesitând refactorizare.
•**Diferență mare între LOC și LOC executabile** → Posibil **cod redundant** sau **instrucțiuni neutilizate**, necesită optimizare.

Cod Generat Automat de LLM

Hiperparametri în LLM-uri

Parametrul	**Descriere**	**Impact asupra codului**
Temperatura	Controlează **nivelul de creativitate** și variație în generarea codului	Valoare mică (aproximativ 0.2) : Cod mai precis și deterministic / Valoare mare (>0.7) : Cod mai creativ, dar posibil mai puțin robust
Top-k sampling	Limitează numărul de tokeni luați în calcul pentru fiecare pas	Valori mici : Cod mai structurat / Valori mari : mai variat, dar apariție posibilă de erori
Top-p sampling (nucleus sampling)	Controlează selecția probabilistică a tokenurilor	Permite generarea unui cod fluent și echilibrat
Fine-tuning	Antrenarea modelului pe seturi de date specifice pentru îmbunătățirea calității codului	Crește adaptabilitatea modelului la cerințele utilizatorului
Cod prompting (includerea exemplarelor specifice în prompt)	Influentează stilul și structura codului generat	Exemple de cod bine scris pot îmbunătăți calitatea rezultatului

Tokenizarea

Numărul de tokeni folosiți într-un model **LLM (Large Language Model)** trebuie să se încadreze într-un interval optim pentru a menține **claritatea, coerența și eficiența** generării de cod. Acest interval depinde de **fereastra de context(numărul maxim de tokeni** ai **modelului. De exemplu GPT-4 opereaza cu max 8000 de tokeni**) și de **complexitatea** codului dorit.

Intervalul optim pentru numărul de tokeni

Scenariu	Număr de tokeni recomandat	Impact asupra codului generat
Cod scurt (funcții simple, fragmente de cod)	**50 - 500**	Cod precis, dar limitat în funcționalitate
Cod mediu (module, structuri de date, clase)	**500 - 2000**	Echilibru între claritate și complexitate
Cod complex (proiecte, aplicații mari)	**2000 - 8000**	Mai multe detalii, dar risc de inconsistență dacă este prea lung

Factori care influențează

- **Tipul de cod generat:** Funcțiile scurte necesită puțini tokeni, aplicațiile complexe au nevoie de un numar considerabil mai mare
- **Fereastra de context a modelului:** Dacă fereastra este prea mică, codul poate deveni incoerent. Un cod bine structurat poate fi generat eficient cu mai puțini tokeni.
- **Bunele practici recomandă:** menținerea numărului de tokeni suficient de mare pentru a asigura claritate și funcționalitatea codului, fără a depăși limita admisa de model.

Top-K Sampling

Top-k sampling este o tehnică utilizată în modelele **LLM** pentru a controla **variația și creativitatea** în generarea de text, inclusiv cod.

- La fiecare pas de generare, modelul prezice **mai mulți tokeni posibili** pentru generarea textului.
- **Top-k** restricționează selecția doar la **cei mai probabili k tokeni**, eliminând opțiunile cu probabilitate foarte mică.
- Acest lucru reduce generarea de tokeni imprevizibili și face output-ul mai **structurat**.

Impact Top-k Sampling asupra Generării de Cod

Pentru o generare de cod optimă, se recomandă **k valori moderate** (50-100) care permite suficientă libertate pentru a explora variante, fără să compromită corectitudinea secvenței

Top-k valoare	Efect asupra codului generat	Avantaje / Dezavantaje
K mic (ex: 10-20)	Cod mai determinist, predictibil	Precizie ridicată, dar posibilă rigiditate
K mediu (ex: 50-100)	Echilibru între diversitate și claritate	Cod coerent, mai creativ / Posibile variații minore
K mare (ex: 500+)	Cod foarte creativ, dar mai puțin precis	Explorare mai liberă / Risc de generare incoerentă

Exemplu

Presupunem că modelul debutează cu promptul: *"Pisica sare peste..."*

P1. Modelul identifică tokenii existenți:

- Pisica
- sare
- peste

P2. Se calculează atenția: modelul determină că "pisica" are o relație puternică cu "sare" și mai slabă cu "peste".

P3. Predictia tokenului următor Modelul analizează probabilitățile pentru posibile continuări:

- gard (probabilitate 85%)
- masă(probabilitate 10%)
- cer(probabilitate 5%)

P4. Alegerea rezultatului final: dacă folosim **Top-k sampling cu k=1**, modelul alege **cel mai probabil token**: "gard". Rezultatul final este: **"Pisica sare peste gard."**

Top-k mic (ex: k=1) → Alegerea cea mai probabilă: gard. **Top-k mare (ex: k=50)** → Mai multe posibilități, Poate alege "masă" sau "cer", ceea ce ar putea suna mai creativ, dar uneori mai puțin realist.

TOP p-sampling

Top-p sampling (numit și **nucleus sampling**) este o metodă de **selectare a tokenilor** în modelele **LLM** care ajută la echilibrarea dintre **precizie și diversitate**:

- Modelul calculează probabilitățile tuturor tokenilor posibili pentru continuarea textului.
- În loc să limiteze selecția la un număr fix (**Top-k**), **Top-p** filtrează **tokenii cei mai relevanți** până când suma probabilităților atinge un prag (ex: 90%).
- Tokenii cu probabilitate mică sunt **ignorați**, menținând o generare mai controlată.

Particularitățile generării de cod prin LLM

- **Controlul coerenței** : Top-p ajută la evitarea tokenilor improbabili care ar putea genera cod incoerent.
- **Echilibru între creativitate și precizie** → Permite explorarea mai multor variante, fără să devină aleatoriu.
- **Reducerea erorilor** → Evită selecția accidentală a tokenilor cu probabilitate scăzută care nu ar fi logici într-un cod.

Comparație Top-k vs Top-p

Metoda	Aspecte funcționale	Impact asupra generării de cod
Top-k	Selectează **cei mai probabili k tokeni**	Cod stabil, dar rigid dacă k este prea mic
Top-p	Alege doar tokenii până când probabilitatea cumulativă atinge p	Cod mai flexibil, elimină alegerile riscante

- **Exemplu:** Dacă un model trebuie să genereze **o funcție Python**, **Top-p** va menține doar tokenii care
- sunt probabil parte din sintaxa corectă, în timp ce **Top-k** ar putea forța anumiți tokeni fără a lua în considerare
- variațiile mai logice.
- **Recomandare:** Pentru generarea de **cod corect, dar variabil**, o valoare **p între 0.85 - 0.95** oferă un
- echilibru bun.

Fine-Tuning pentru Generarea de Cod

- **Fine-tuning** este procesul prin care un model **LLM** este adaptat pentru a genera **cod optimizat** în funcție de cerințele specifice ale unui utilizator sau domeniu.
- **Îmbunătățirea preciziei** modelului: Modelul învață sintaxa și structura specifică a unui limbaj de programare.
- **Reducerea erorilor**: Se evită generarea de cod incoerent sau redundant.
- **Adaptare la convențiile de cod** : Se pot personaliza stilurile de programare și practicile standard.
- **Creșterea eficienței :** Modelul devine mai rapid în generarea codului potrivit pentru un proiect.

Ciclul de viață pentru Fine –Tuninng

- **Colectarea datelor** : Se colectează exemple de cod relevant pentru antrenare.
- **Preprocesarea datelor** : Se ajustează codul și se elimină redundanțele.
- **Alegerea hiperparametrilor :** Se setează valori optime pentru învățare (ex. temperatura, top-p, context window etc.).
- **Antrenarea modelului** : Se rulează procesul de fine-tuning pe setul de date pregătit.
- **Validarea și testarea** : Se verifică performanța modelului pentru a vedea cât de bine generează cod corect și eficient.

Fine-Tuning Exemple

- **Cod optimizat pentru un limbaj specific**: Python, JavaScript, C++, etc.
- **Îmbunătățirea generării de funcții complexe** : Algoritmi eficienți și structuri modulare.
- **Generarea de cod conform unui stil prestabilit** : Convenții de indentare, documentare, standarde de echipă.

Q & A