Tema 1: Regresie Liniara

Deadline: `12 aprilie 2024`

Schelet de cod: t1_v2.zip

Schelet actualizat 24 martie - adaugat teste pentru Regresie.

Cerinta actualizata 26 martie - update cerinta `split` & schelet actualizat pentru testarea lui `split`.

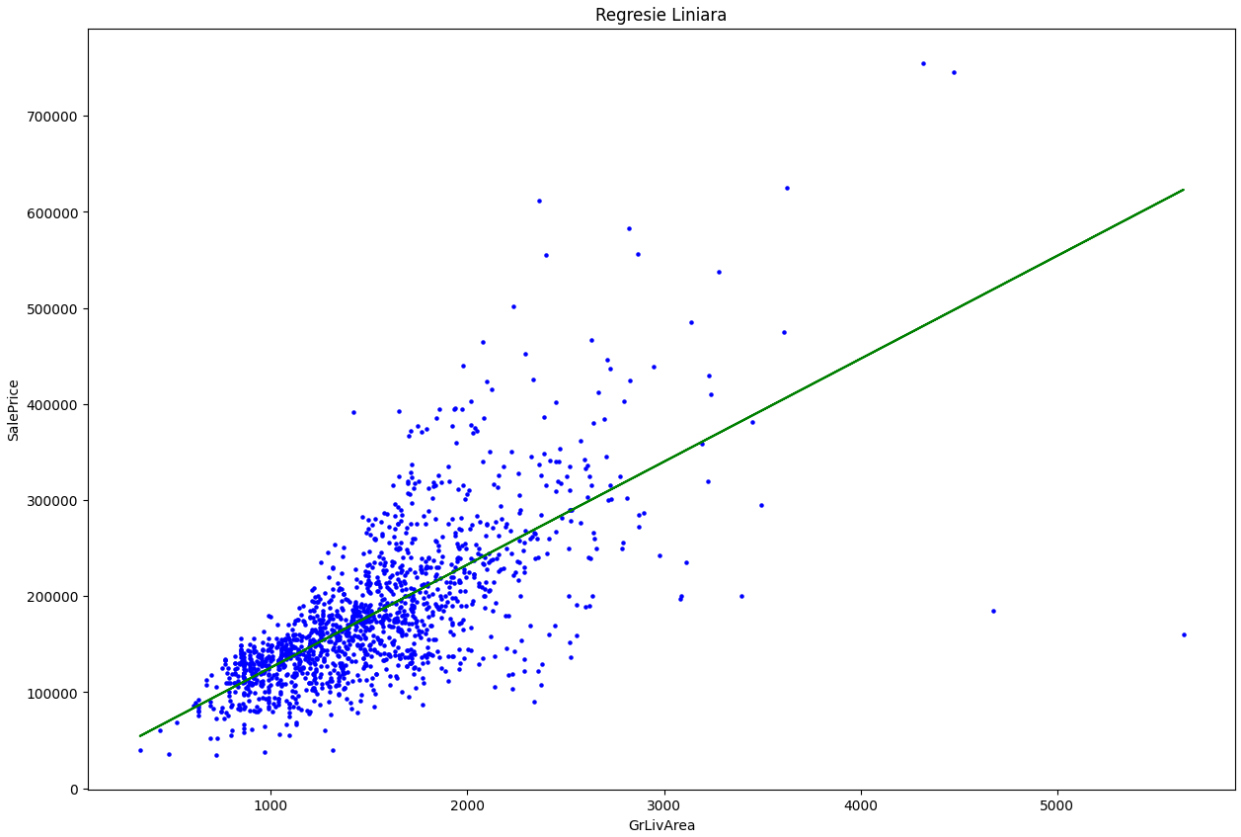
Folosiți un stil de programare funcțional.
NU se vor accepta:

- Efecte laterale (de exemplu modificarea parametrilor dați ca input la funcție)
- `var` (`val` este ok!)
- `REZOLVARILE IDENTICE CU CELE DIN CHECKER NU VOR FI PUNCTATE` ⇒ Cateva dintre teste se bazeaza pe o implementare in stil procedural a cerintei. Implementarile voastre trebuie sa fie functionale, deci folositi in rezolvarea voastra oriunde este posibil functii de ordin superior (e.g. foldRight, foldLeft, map, zip).

Intro - ce este Regresia Liniara?

Imaginati-va ca avem o multime de puncte pe o foaie de hartie, fiecare punct avand o valoare pe axa orizontala si una pe axa verticala. **Regresia liniara** este o tehnica de Machine Learning care produce o dreapta cat mai potrivita printre aceste puncte, astfel incat suma distantelor dintre puncte si linia trasata sa fie minima. Pentru 2 axe (x si y), o regresie liniara consta intr-o dreapta descrisa de ecuația $y=a*x+b$ unde a si b sunt constante ce trebuie determinate de voi pentru a minimiza distanta dintre puncte la dreapta.

Vom numi coordonatele pe axa x a punctelor din poza de mai jos **input**. Valoarea de pe axa y a fiecarui punct o vom numi **valoarea reala**. In plus, valorile $y(x)$ de pe dreapta determinata se vor numi **predictii**. Cu aceasta taxonomie, putem spune ca regresia liniara estimeaza o dreapta, astfel incat, suma tuturor erorilor predictiilor (diferenta intre valoarea prezisa si valoarea reala) sa fie **minima**.



Ecuatia dreptei din imagine este: y=118.06894201605218∗x+0.2687269842929364y=118.06894201605218∗x+0.2687269842929364 .
In graficul de mai sus sunt reprezentate preturile de vanzare ale unor proprietati (majoritatea case cu mai multe etaje), in raport cu suprafata totala locuibila a acestora. Axa xx reprezinta suprafata, iar yy reprezinta pretul. Privind graficul *de la distanta* se poate observa o dependenta **liniara** intre preturi si suprafete. Putem **estima** pretul unei proprietati ca o functie/dreapta y=a∗x+by=a∗x+b , unde xx reprezinta suprafata, iar yy reprezinta pretul. O astfel de dreapta este ilustrata cu verde in graficul de mai sus si puteti observa ca aceasta estimeaza foarte bine unele preturi, cele ale proprietatilor sub 2000 mp, si mai putin bine pe cele cu suprafata mai mare.
In general, pretul unei case nu depinde doar de suprafata total, ci si de alte **atribute/feature-uri**. Astfel, daca luam si alte atribute ale locuintei in calcul, precum anul in care a fost construita, vom putea obtine o estimare de preț mai buna. In acest caz, estimarea pretului se face folosind o functie liniara cu **doua variabile**: y=a∗x0+b∗x1+cy=a∗x0+b∗x1+c , unde:

- x0x0 si x1x1 reprezinta suprafata totala, respectiv anul construcției
- aa , bb si cc reprezinta parametrii modelului, pe care, folosind regresia, ii calculam astfel incat sa minimizeze suma tuturor erorilor predictiilor.

Puteti vizualiza reprezentarea grafica a acestei situatii cu acest fisier:3d_graph.zip.

In general, regresia poate fi implementata cu un singur atribut, cu doua sau si cu mai multe, in functie de informatiile disponibile despre procesul ce se doreste a fi prezis. Dincolo de aceste informatii, alte cunostinte despre regresie nu sunt necesare pentru implementarea acestei teme.

1. Seturi de date

Vom incepe cu procesarea datelor pe care le vom folosi pentru construirea estimarii.

1.1. Citirea din CSV - 15p

Datele sunt in format CSV, in care fiecare coloana reprezinta un atribut sau feature, iar valorile de pe fiecare linie sunt separate prin virgule. Mai jos se găsește un exemplu de fisier CSV avand un singur feature numeric (suprafata locuita totala) si rezultatul asteptat pe ultima coloana (pretul).

```
GrLivArea,SalePrice
100,50000
200,100000
210,100500
300,153000
```

Vrem sa extragem datele dintr-un CSV intr-un obiect de tip `Dataset`, care va mentine intern o structura **tabelara** (matriceala) de tip `List[List[String]]`.

1.1.0. Pentru inceput, definiti marimea unui `Dataset`, care sunt liniile sale si care este header-ul cu numele de coloane.

```
def size = ???
def getRows: List[List[String]] = ???
def getHeader: List[String] = ???
```

1.1.1. Implementati metoda `apply` din obiectul companion`Dataset`. Aceasta construieste o instanta a clasei `Dataset` citind datele dintr-un fisier CSV al carui cale este data ca parametru.

```
apply(csv_filename: String): Dataset = ???
```

1.1.2. In plus, implementati reprezentarea ca `String` al unei instante de `Dataset`:

```
override def toString: String = ???
```

Hint: Aceasta functie va va mai trebui poate, ar fi indicat sa o implementati separat, intr-un singur loc, de exemplu: intr-un nou fisier cu functii ajutatoare.

1.1.3. In continuare, completati metoda `apply` de mai jos, care supraincarca metoda anterioara:

```
def apply(ds: List[List[String]]): Dataset = ???
```

Pentru citirea din fisier recomandam sa folositi clasa `Source` din biblioteca Scala si metodele `fromFile` si `getLines`. `Source.fromFile(filename).getLines` va intoarce o lista cu toate liniile din fisierul cu numele `filename`.

1.2. Selectare atribute/feature-uri - 15p

In implementarea acestei teme vom folosi un set de date real, in care au fost documentate numeroase caracteristici ale unor proprietati, inclusiv pretul lor de vanzare. Puteti citi aici o scurta descriere a fiecarei coloane din `houseds.csv`.

In cadrul temei, nu vom folosi toate coloanele pentru regresie (desi ar fi posibil, inclusiv pentru cele care nu sunt numerice).

Implementati metodele `selectColumn` si `selectColumns` din clasa `Dataset`, care vor intoarce un set de date restrans, ce contine doar coloanele al caror nume este primit ca parametru.

```
def selectColumn(col: String): Dataset = ???

def selectColumns(cols: List[String]): Dataset = ???
```

Pentru fiecare coloana din lista furnizata ca parametru, exact in aceasta ordine si indiferent de duplicate, veti extrage coloana din setul de date corespunzatoare.

Pentru implementarea acestor functii, folositi **functii de ordin superior** (e.g. `foldRight`, `foldLeft`, `map`, `zip`).

1.3. Impartire Dataset - 10p

Toate metodele de Machine Learning:

- folosesc doar o parte din datele disponibile pentru **estimarea** parametrilor (proces numit **antrenare**).
- un procent mai mic (~20%) din date este rezervat pentru **a evalua** performanta modelului invatat, model care in cazul nostru este regresia liniara. Este important ca aceste date sa fie **noi** (ne-*vazute* in timpul antrenarii), pentru ca evaluarea sa arate cat mai clar cum se comporta modelul pe date ce nu au facut parte din antrenare.

Implementati metoda:

```
def split(percentage: Double): (Dataset, Dataset) = ???
```

care imparte setul de date in doua seturi. Valoarea `percentage` este intre `0` si `0.5` si reprezinta procentul din dataset ce va fi pastrat pentru evaluare, din totalul de intrari ale dataset-ului. Metoda va intoarce o pereche de dataset-uri: unul mai mare (numit “de antrenare”) si unul mai mic (“de testare/validare”). Pentru aceasta impartire, urmati pasii urmatori:

1. Sortati setul de date crescator dupa prima coloana
2. Pentru fiecare $1/\text{percentage}-1$1/*percentage*−1 intrari consecutive din dataset-ul sortat, adaugati o intrare in dataset-ul de evaluare, iar restul - in cel de training.

Aveti grija la randul de cap de tabel. Aceasta nu este considerat o intrare si trebuie sa apara in ambele dataseturi construite.

2. Operatii cu matrici

Pentru ca regresia liniara functioneaza doar cu date numerice, avem nevoie sa transformam un `Dataset`, ale carui campuri sunt de tip `String`, intr-o matrice de `Double`, daca se poate. Matricile vor reprezenta intern datele ca `Option[List[List[Double]]]` tocmai pentru a ilustra ideea de aparitie a unei erori in timpul conversiei sau al altor operatii.

`Option` este un TDA deja existent in Scala si reprezinta o valoare care s-ar putea sa nu existe. Option are 2 constructori:

- `None` - valoarea nu exista
- `Some(x)` - valoarea exista si este `x`

Vi-l puteti imagina implementat ca mai jos:

```
trait Option {}

case class Some(value: List[List[Double]]) extends Option{}

case object None extends Option{}
```

Vom folosi `Option` pentru a trata cazurile de eroare ce pot aparea la operatiile de inmultire si scadere. Vom spune ca o matrice ce contine un None contine o “eroare”.

Pentru a simplifica lucrul cu acest tip de date, recomandam sa folostiti `pattern matching`.

Toate operatiile (functiile de ordin superior - foldRight, foldLeft, map, zip etc.) se vor efectua pe **matricea interna**, nu pe TDA-ul Option!

Puteti presupune ca orice matrice primita este de forma dreptungiulara (are cel putin o linie si toate liniile sale au aceeasi dimensiune).

2.1. Conversie din dataset

2.1.0. Descrieti lungimea si latimea unui `Matrix`.

```
def height: Option[Int] = ???

def width: Option[Int] = ???
```

2.1.1. Pentru a realiza conversia din `Dataset` in `Matrix`, implementati functia de mai jos, tinand cont de urmatoarele indicatii:

- Ignorati linia de cap de tabel.
- Folositi `toDouble` pentru conversia `String` → `Double`

```
 def apply(dataset: Dataset): Matrix = ???
```

2.2. Transpunere - 5p

Definiti operatia de transpunere a unei matrici. Daca matricea contine o eroare (`Matrix(None)`), aceasta metoda va intoarce tot o eroare.

```
def transpose: Matrix = ???
```

2.3. Map - 5p

Aplicati o transfomare pe fiecare element al matricii urmand principiul de funtionare al functiei map cunoscuta de la liste. Daca matricea contine o eroare (`Matrix(None)`), aceasta metoda va intoarce tot o eroare.

```
def map(f: Double => Double): Matrix = ???
```

2.4. Scadere - 5p

Realizati operatia de scadere (element cu element) intre doua matrici. Daca vreuna din cele doua matrici contine erori sau daca scaderea nu se poate efectua (dimensiunile matricilor nu sunt compatibile), aceasta metoda va intoarce o eroare.

```
def -(other: Matrix): Matrix = ???
```

2.5. Inmultire - 5p

Implementati operatia de inmultire matriciala: considerand inmultirea `A * B = C`, elementul de la linia `i` coloana `j` din `C` se obtine prin inmultirea element cu element a liniei `i` din `A` cu coloana `j` din `B`, si insumarea valorilor obtinute. Daca vreuna din cele doua matrici contine erori sau daca inmultirea nu se poate efectua (dimensiunile matricilor nu sunt compatibile), aceasta metoda va intoarce o eroare.

```
def *(other: Matrix): Matrix = ???
```

2.6. Adaugare de coloana - 5p

In general, cand dorim sa calculam o regresie, trebuie sa determinam un termen constant b din ecuatia $y=a\cdot x+b$. Insa, din punct de vedere al implementarii, ne este mai usor sa calculam coeficientii a si b daca ecuatia este adaptata la forma $y=a\cdot x+b\cdot C$, ceea ce ne permite sa folosim operatii de inmultire matriceala. In acest context, C este o coloana care contine aceeasi valoare constanta pe toate randurile sale.
De aceea, vrem sa implementam functia care adauga o coloana de valoare constanta (egala cu `x`) la dreapta matricii. Daca matricea contine o eroare (`Matrix(None)`), aceasta metoda va intoarce tot o eroare.

```
def ++(x: Double): Matrix = ???
```

3. Regresia Liniara

Cel mai simplu (si des intalnit) mod de a gasi parametrii din ecuatia dreptei care descriu regresia aleasa este algoritmul `Gradient Descent`, ce consta in urmatorii pasi:

1. **Incarcarea si selectia datelor:**
   - Se porneste cu un set de date care contine multiple coloane.
   - Fiecare rand reprezinta o inregistrare, iar coloanele reprezinta diferite informatii despre aceste inregistrari.
   - Una dintre aceste coloane este variabila pe care incercam sa o estimam (ex: pretul unei case), iar restul sunt informatii pe care le folosim pentru a face aceasta estimare (ex: marimea casei, numarul de camere, etc.).
   - Pentru acest pas veti folosi functia `selectColumns` din `Dataset` pentru a pastra din setul de date doar coloanele de interes.
   - In plus, separati setul de date obtinut in bucati mai mici, unul de antrenare - folosit pentru a determina parametrii regresiei, si unul de validare - folosit pentru a calcula cat de buna este regresia obtinuta.
2. **Crearea matricei cu date de intrare (X):**
   - Vom folosi setul de date de antrenare.
   - Daca avem n locuinte, fiecare rand din matricea `X` va contine valorile atributelor acestor locuinte, pe care le-am extras la punctul anterior.
   - Vom adauga o coloana aditionala cu valoarea 1 la sfarsitul lui X, pentru a include un termen liber. Acest lucru ne va face calculele mai usor de realizat folosing inmultiri de matrici.
   - Daca avem `m` locuinte si `n` atribute, `X` va fi o matrice de marime $\mathrm{m}*(\mathrm{n}+1) m*(n+1)$
   - Folositi functia `++` din `Dataset`.
3. **Initializarea coeficientilor regresiei / ipotezei (W):**
   - Se initializeaza un vector de parametri `W` cu dimensiunea $(\mathrm{n}+1)*1 (n+1)*1$ (atatea coloane cat are `X`). Fiecare element din `W` reprezinta coeficientul asociat fiecarei atribut numeric din X. Initial, toate valorile din `W` sunt setate la 0. Ulterior ele vor fi actualizate de algoritmul de Gradient Descent pentru a exprima ecuatia dreptei ce descrie regresia noastra.
4. **Gradient Descent:** Pentru un numar predefinit de pasi (`gradient_descent_steps`), se executa urmatorii sub-pasi:
   a. **Calculul estimarilor:** Folosind atributele fiecarei locuinte si ipoteza de la pasul curent (coeficientii asociati fiecarui atribut), vom inmulti matricea `X` de dimensiune $\mathrm{m}*(\mathrm{n}+1) m*(n+1)$ cu vectorul de coeficienti `W` de dimensiune $(\mathrm{n}+1)*1 (n+1)*1$, rezultand un **vector de estimari** de dimensiune $\mathrm{m}*1 m*1$. Acest vector de estimari reprezinta practic pretul prezis de regresia noastra pentru fiecare din cele m locuinte din setul de date. Cu alte cuvinte, daca `W` are coeficientii $\mathrm{W}_0,\mathrm{W}_1,...,\mathrm{W}_\mathrm{n} W0, W1, \ldots, Wn$ iar o locuinta are atributele $1,\mathrm{X}_1,\mathrm{X}_2,...,\mathrm{X}_\mathrm{n} 1, X1, X2, \ldots, Xn$, noi vom calcula pretul prezis ca fiind $\mathrm{W}_0+\mathrm{W}_1*\mathrm{X}_1+\mathrm{W}_2*\mathrm{X}_2+...+\mathrm{W}_\mathrm{n}*\mathrm{X}_\mathrm{n} W0+W1*X1+W2*X2+\ldots+Wn*Xn$.
   b. **Calculul erorii:** In continuare, vrem sa facem astfel incat pretul prezis sa se apropie cat mai mult cu pretul real al locuintelor, definind astfel o functie de eroare egala cu diferenta dintre pretul prezis si cel real (preturile prezise sunt valorile din vectorul de estimari). Pretul real pentru toate locuintele se va retine ca un vector `Y` cu dimensiunea $\mathrm{m}*1 m*1$. Astfel eroarea va avea, de asemenea, dimensiunea $\mathrm{m}*1 m*1$.

c. **Calculul gradientului:** Gradientul reprezinta directia (crestem sau scadem) si marimea ajustarii necesare pentru coeficientii `W`, pentru a reduce eroarea. Se calculeaza inmultind transpusa matricei `X` (de dimensiune $(n+1)*m(n+1)*m$) cu vectorul de eroare (de dimensiune $m*1m*1$), si apoi se imparte fiecare element la m pentru a obtine media aritmetica. Rezultatul este un vector de dimensiune $(n+1)*1(n+1)*1$.
d. **Actualizarea ipotezei (W):** Se actualizeaza `W` scazand produsul dintre gradient si un pas de invatare (alpha, un scalar), din valorile curente ale lui `W`. Acest pas determina cat de repede invatam sau ajustam parametrii - influentand viteza de convergere la valorile optime. Noua valoare a lui `W` va fi folosita in urmatoarea iteratie a algoritmului.

5. Acum ca avem coeficientii regresiei, pentru ca am calculat matricea `W` dupa `gradient_descent_steps` pasi, vrem sa vedem cat de buna este estimarea noastra. Acum vom lua setul de date de validare si vom calcula predictiile regresiei noastre pentru fiecare locuinta. In continuare, vom calcula eroarea ca media aritmetica intre diferentei dintre pretul prezis si cel real pentru fiecare locuinta din acest set de date de validare.
6. Vom intoarce ipoteza finala `W` si eroarea pe setul de validare.

Click to display ↘

3.1. Regresia pentru o lista de atribute - 30p

In fisierul TestRegression exista 10 teste unitare. Fiecare valorează 3p din punctul aferent temi. Cu alte cuvinte cele 10 teste se scaleaza la 30p.

Implementati functionalitatea metodei `regression` in interiorul clasei `Regression`. Aceasta metoda trebuie sa execute urmatorii pasi, presupunand ca lucram cu `n` atribute selectate pentru a realiza regresia liniara. Matricea de intrare `X` va contine `n + 1` coloane: `n` coloane pentru atributele selectate si o coloana suplimentara care contine constanta `1` pentru a gestiona termenul liber al regresiei.

Parametrii metodei `regression` sunt:

- dataset_file - Calea catre fisierul care contine setul de date.
- attribute_columns - Lista cu numele coloanelor care vor fi folosite ca atribute.
- value_column - Numele coloanei care va fi folosita ca variabila tinta.
- test_percentage - Procentul de impartire intre setul de date de antrenare si cel de validare.
- alpha - Rata de invatare utilizata in actualizarea parametrilor modelului.
- gradient_descent_steps - Numarul de iteratii pentru care algoritmul Gradient Descent va fi executat.

Metoda `regression` trebuie sa realizeze urmatoarele actiuni:

1. impartiti setul de date initial in doua subseturi: de antrenare si de validare, folosind `test_percentage`.
2. Preprocesati setul de antrenare pentru a include coloana de 1-uri, generand astfel matricea `X` de dimensiune $m*(n+1)m*(n+1)$, unde `m` este numarul de linii din setul de antrenare.
3. Initializati vectorul de parametri `W` cu dimensiuni $(n+1)*1(n+1)*1$, toate valorile fiind setate initial la `0`.
4. Aplicati algoritmul Gradient Descent pe setul de antrenare pentru numarul specificat de `steps`, ajustand parametrii `W` folosind rata de invatare `alpha`.

5. Dupa finalizarea pasilor de Gradient Descent, folositi `W` pentru a genera predictii pe setul de validare.
6. Calculati eroarea totala ca media aritmetica a diferentelor intre valorile prezise si valorile reale din setul de validare.
7. Intoarceti un tuple format din vectorul de parametri `W` ajustat si suma erorilor calculate pentru setul de validare.

Asigurati-va ca functia gestioneaza corespunzator preprocesarea datelor si impartirea in seturi de antrenare si validare, precum si calculul precis al gradientului si actualizarea parametrilor conform algoritmului Gradient Descent.

```
def regression(
  dataset_file: String,
  attribute_columns: List[String],
  value_column: String,
  test_percentage: Double,
  alpha: Double,
  gradient_descent_steps: Int
): (Matrix, Double) = ???
```

3.2. Plotting - 5p

Pornind de la fisierul `houseds.csv` realizati o regresie pe baza atributelor (coloanelor) `GrLivArea` si `YearBuilt` , incercand sa preziceti valoarea coloanei `SalePrice`. Reprezentati grafic, folosind gnuplot, planul de regresie si sample-urile.

Pentru a genera plot-ul, folositi urmatorul script, inlocuind valorile `A`,`B`,`C` cu valorile corespunzatoare obtinute din regresie:

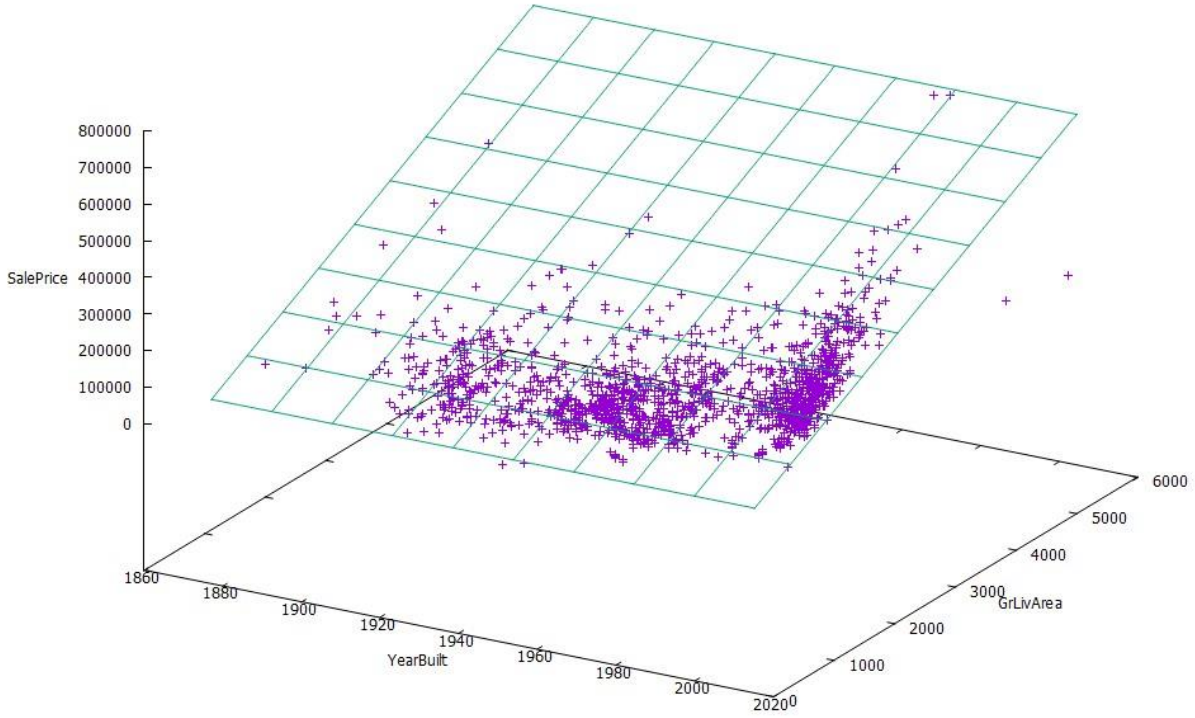
plot.plt

```
set datafile separator ","
set xlabel "YearBuilt"
set ylabel "GrLivArea"
set zlabel "SalePrice" offset -5,0,0
splot 'datasets/houseds.csv' using "YearBuilt":"GrLivArea":"SalePrice" with points, A + B * x + C * y
```

Folositi valoarea `0.1` (10%) pentru split-ul test-train. Recomandam sa folositi un alpha de `1e-7` si `10000` de pasi de antrenare.

Pentru a rula scripul, folositi comanda `load '<path catre script>'`

Rezultatul va fi similar cu imaginea de mai jos:



Testare

In cadrul acestei teme, testarea se va face folosind `Scalacheck`, o biblioteca de property-based testing si `munit`, o biblioteca de teste unitare. Vom verifica implementarile de Dataset si Matrix folosind Scalacheck, iar Regression va fi testat cu munit.

Munit este o biblioteca de testare unitara pentru Scala.Testele unitare sunt o metoda de testare a unitatilor individuale de cod, cum ar fi functii, metode sau clase, in izolare de alte parti ale aplicatiei. Scopul principal al testelor unitare este de a verifica daca unitatile individuale de cod functioneaza conform asteptarilor specificate. Aceste teste sunt scrise de catre dezvoltatori pentru a valida comportamentul corect al codului lor si pentru a identifica eventualele erori sau bug-uri in mod eficient. Testele unitare preiau un input predefinit si verifica daca rezultatul functiilor implementate este acelasi cu cel asteptat.

Scalacheck este o biblioteca pentru Scala care permite testarea automata a codului. Scalacheck genereaza o suita de date de testare random si verifica daca proprietatile specificate de utilizator sunt adevarate pentru acele date. Practic, aceasta este o metoda de verificare a corectitudinii codului semi-formala, in sensul ca incearca inputuri generice de orice fel, dar nu acopera intreaga plaja de inputuri posibile. Insa, faptul ca inputurile sunt aleaatorii face foarte probabil ca bug-urile sa fie detectate.

Property-based testing este un stil de testare care se bazeaza pe proprietati care ar trebui sa fie adevarate pentru orice input valid. In comparatie cu testarea unitara, care se bazeaza pe input-uri

fixe, property-based testing genereaza input-uri random si verifica daca proprietatile sunt adevarate pentru acele input-uri. Natura randomizata a input-urilor face ca property-based testing sa fie mai puternic decat testarea unitara, deoarece acopera un spatiu mai mare de input-uri si poate detecta bug-uri care nu ar fi fost detectate de testarea unitara.

Scalacheck va genera input-uri random si va rula testele pentru acele input-uri. Daca un test esueaza, Scalacheck va afisa input-ul pentru care testul a esuat, ceea ce face debugging-ul mai usor. In plus, Scalacheck ofera suport pentru shrinking, care reduce input-ul generat pentru a gasi un input mai mic care esueaza testul, acesta fiind mai usor de inteles si de debug.

In IntelliJ, in fisierul build.sbt se afla referinta:

```
libraryDependencies += "org.scalacheck" %% "scalacheck" % "1.14.1" % "test"
libraryDependencies += "org.scalameta" %% "munit" % "0.7.29" % Test
```

Aceasta este suficienta pentru a va lasa sa rulati fiserele de testare din IDE.

Daca folosiți terminalul:

- PropertiesDatabase si PropertiesMatrix se rulează in mod obisnuit, având o metoda main.
- Pentru a rula TestRegression, folositi comanda următoare in radacina scheletului:

```
sbt test
```

Sunteti liberi (si incurajati) sa adaugati si alte teste daca vi se par utile.

Checkerul nu acorda punctajul pentru task-ul 3.2. Acesta va fi acordat manual pe baza corectitudinii scriptului de plot.

Submisie arhiva

Veti incarca pe moodle o arhiva ce contine, in radacina acesteia, folderul `src` al proiectului vostru, fisierul `build.sbt`, script-ul de gnuplot, denumit `plot.plt`, si un fisier text, intitulat `ID.txt` ce contine o singura linie, si anume id-ul vostru anonim (pe care il puteti gasi pe moodle la assignment-ul `tokenID`).

Exemplu structura arhiva:

```
archive.zip
|-src/
| |-main/
| | |-scala/
| | | | - ... <fisierele cu sursa scala>
|-build.sbt
```

```
|-ID.txt
|-plot.plt
```