Rešavanje problema skupa atributa uz pomoć genetskog algoritma

Anja Miletić

8. septembar 2021.

Sadržaj

1 Uvod

Problem biranja najmanjeg podskupa atributa (eng. Minimum feature subset selection problem) je bitan u istraživanju podataka i mašinskom učenju. Količina podataka je sve veća, pa su tehnike čišćenja, odnosno smanjivanja obimnosti podataka važnije. U slučaju najmanjeg podskupa atributa, cilj je smanjiti dimenzionalnost podataka bez gubljenja tačnosti klasifikacije, radi bržeg treniranja neuronskih mreža.

2 Metode

Dati problem je NP težak, i teško ga je aproksimirati [1]. Postoje različiti pristupi rešavanju problema. Nabrojimo neke [2]:

- selekcija pojedinačnih atributa koristeći metrike koje se odnose na važnost tih atributa. Naravno ova metoda je brža i deluje jednostavno, ali ne uzima u obzir odnos između atributa, samim tim ne daje najbolje rezultate.
- potpuna pretraga prostora rešenja. Od svih valjanih podskupova atributa izabrati onaj koji maksimizuje tačnost. Ova metoda je optimalna - od svih mogućih rešenja ona će dati najmanji podskup. Zbog velikog broja mogućih rešenja, potpuna pretraga se može koristiti samo ako je ukupan broj atributa, odnosno ukupan broj mogućih podskupova mali
- heurističke metode pretrage. Sequential Forward Selection (SFS) i Sequential Backward Selection (SBS) algoritmi koriste heuristiku: 'Najbolji atribut za dodavanje u svakom koraku je atribut koji treba izabrati'. Ova heuristika ne uzima obzir interakciju izmedju atributa.
- Relief algoritam. Baziran na težinama atributa, algoritam prepoznaje one atribute koji su statistički bitni, proučavajuci razlike izmedju vrednosti podataka. Algoritam je polinomijalne složenosti.

3 Predlog rešenja korišćenjem genetskog algoritma

Genetski algoritmi pripadaju porodici optimizacionih algoritama koji traže globalni minimum fitnes funkcije. U opstem slučaju, to uključuje četiri koraka: [3]

- evaluacija: bira se pseudo-slučajan set početne populacije. Izračunava se fitnes svake jedinke, zatim se one sortiraju na osnovu fitnesa.
- reprodukcija: biraju se najbolje jedinke koje se čuvaju u narednoj generaciji. Ove jedinke se zovu elitna deca.
- rekombinacija: biraju se jedinke koje će se ukrstiti da bi se napravile jedinke za sledeću generaciju. U ovom koraku pokušavamo da sačuvamo najbolje gene i kombinujemo ih da napravimo još bolje jedinke.

- mutacija: mali procenat populacije prolazi kroz mutaciju (izmenu dela koda). Ovaj korak je bitan da bi se izbeglo zaglavljivanje u lokalnom minimumu.

3.1 Postavka rešenja

Koristićemo genetski algoritam da nađemo najmanji skup atributa sa najvećom preciznošću. Jedinke će biti kodirane binarnim nizom dužine m, gde je m ukupan broj atributa našeg seta podataka. Ako je na i-toj poziciji u nizu vrednost `True`, onda u podskup atributa ulazi i-ti atribut. Fitnes jedinke predstavlja preciznost klasifikacije modela koji je učen nad podacima koji sadrze podskup atributa predstavljen kodom jedinke. Koristićemo sekvencijalni model `keras` biblioteke koji je optimizovan za probleme klasifikacije.

3.1.1 Podaci

Koristićemo podatke sa košarkaskih utakmica NBA sezone 2020-21. Atributi se odnose na razne statističke parametre koji se prate tokom utakmice, a zabeleženi su tokom prvog poluvremena. Klasifikikaciju radimo u odnosu na to da li je tim pobedio ili izgubio, pa zbog toga ne gledamo parametre sa kraja utakmice. Nakon čišćenja podataka ostaje nam 21 numerički atribut, kao i kolona `W\\L` koju mapiramo kao `W->1, L->0`.

3.1.2 Algoritam

U nastavku sledi kod za određivanje fitnesa jedinke:

```
def getModelAccuracy(featuresUsed, data):
    # get array of True value indexes, eg [True, False, True] -> [0, 2]
    remainingFeatures = [x for x in range(len(featuresUsed)) if featuresUsed[x]]
    data_final = data.iloc[:, remainingFeatures]

    # create model
    X_train, X_test, y_train, y_test = train_test_split(data_final__,
        results_final, test_size=0.33, random_state=7, stratify=results_final)
    scaler = StandardScaler()
    scaler.fit(X_train)
    X_train = scaler.transform(X_train)
    X_test = scaler.transform(X_test)

    model = Sequential()
    model.add(Dense(input_dim=X_train.shape[1], units=500, activation='relu',
        kernel_constraint=unit_norm()))
    model.add(Dropout(rate=0.2))
    model.add(Dense(units=100, activation='relu',
        kernel_constraint=unit_norm()))
    model.add(Dense(units=1, activation='sigmoid'))
    model.compile(optimizer='adam', loss='binary_crossentropy',
        metrics=['accuracy'])
```

```
    history = model.fit(X_train, y_train, batch_size=64, epochs=20, verbose=1,
        validation_split=0.3)
    return history.history['val_accuracy'][-1]
```

Koristimo metriku `val_accuracy`, tačnost klasifikacije validacionog skupa, da bismo izbegli slučajeve gde je `accuracy = 100.0` zbog preprilagođavanja.
Klasa `Individual` sadrži podatke o jedinki, kao i metode jedinke koje koristimo tokom algoritma:

```
class Individual():
    def __init__(self, numResources):
        # code is a binary array where 0 represents the absence of a feature
        self.code = [random.random() < 0.5 for _ in range(numResources)]
        self.correctNonFeasible()

        self.fitness = self.calculateFitness()

    def __lt__(self, other):
        if (self.fitness == other.fitness):
            # the smaller the code, the better (less features)
            return len([x for x in self.code if x]) < len([x for x in
                other.code if x])

        # a better individual has a higher accuracy
        return self.fitness > other.fitness

    def invert(self):
        i = random.randrange(len(self.code))
        self.code[i] = not self.code[i]
        if self.isFeasible():
            return i
        return -1

    def isFeasible(self):
        for c in self.code:
            if c:
                return True
        return False

    def correctNonFeasible(self):
        for c in self.code:
            if c:
                return
        # at least one feature must be present
        index = random.randrange(0, len(self.code))
        self.code[index] = True

    def calculateFitness(self):
```

```
        return getModelAccuracy(self.code)
```

Ukoliko je fitnes dve jedinke isti, nama je vrednija ona sa manjim skupom atributa. Skup atributa ne sme da bude prazan, tako da u tom slučaju biramo nasumično jedan atribut koji će pripadati skupu.
Uvodimo pomoćne funkcije selekcije, ukrštanja i mutacije:

```
def selection(population):
    TOURNAMENT_SIZE = 6
    maxAccuracy = float('-inf')
    bestIndex = -1

    for i in range(TOURNAMENT_SIZE):
        index = random.randrange(len(population))
        if population[index].fitness > maxAccuracy:
            maxAccuracy = population[index].fitness
            bestIndex = index

    return bestIndex

def crossover(parent1, parent2, child1, child2):
    breakpoint = random.randrange(0, len(parent1.code))

    child1.code[:breakpoint] = parent1.code[:breakpoint]
    child2.code[:breakpoint] = parent2.code[:breakpoint]

    child1.code[breakpoint:] = parent2.code[breakpoint:]
    child2.code[breakpoint:] = parent1.code[breakpoint:]

    child1.correctNonFeasible()
    child2.correctNonFeasible()

def mutation(individual):
    MUTATION_PROB = 0.05
    for i in range(len(individual.code)):
        if random.random() < MUTATION_PROB:
            individual.code[i] = not individual.code[i]

    individual.correctNonFeasible()
```

Koristićemo turnirsku selekciju i jednopoziciono ukrštanje. Slede definicije samog algortima:

```
def genAlgWithoutSimulatedAnnealing():
    POPULATION_SIZE = 20
    numResources = data_final.shape[1]
    population = [Individual(numResources) for i in range(POPULATION_SIZE)]
    newPopulation = [Individual(numResources) for i in range(POPULATION_SIZE)]
```

```
    ELITISM_SIZE = int(0.3 * POPULATION_SIZE)
    MAX_ITER = 30
    for i in range(MAX_ITER):
        population.sort()
        newPopulation[:ELITISM_SIZE] = population[:ELITISM_SIZE]
        for j in range(ELITISM_SIZE, POPULATION_SIZE-1, 2):
            parent1Index = selection(population)
            parent2Index = selection(population)

            crossover(population[parent1Index], population[parent2Index],
                newPopulation[j], newPopulation[j+1])

            mutation(newPopulation[j])
            mutation(newPopulation[j+1])

            newPopulation[j].fitness = newPopulation[j].calculateFitness()
            newPopulation[j + 1].fitness = newPopulation[j +
                1].calculateFitness()

        population = newPopulation

    bestIndividual = max(population, key=lambda x: x.fitness)
    print('Solution: {}, fitness: {}'.format(bestIndividual.code,
        bestIndividual.fitness))
```

3.2 Varijacije

3.2.1 Simulirano kaljenje

Pored običnog algoritma implementiraćemo i algoritam sa simuliranim kaljenjem. Simulirano kaljenje je još jedna tehnika koja omogućuje izlazak iz lokalnog minimuma, tako što se prihvata gore rešenje, sa verovatnoćom obrnuto-proporcionalnom broju iteracija.

```
def simulatedAnnealing(individual, iters):
    for i in range(iters):
        j = individual.invert()
        if j < 0:
            continue
        newFitness = individual.calculateFitness()

        if newFitness > individual.fitness:
            individual.fitness = newFitness
        else:
            p = 1.0 / (i + 1) ** 0.5
            q = random.uniform(0, 1)
            if p < q:
                individual.fitness = newFitness
```

```
            else:
                individual.code[j] = not individual.code[j]

def genAlgSimulatedAnnealing():
    POPULATION_SIZE = 20
    numResources = data_final.shape[1]
    population = [Individual(numResources) for i in range(POPULATION_SIZE)]
    newPopulation = [Individual(numResources) for i in range(POPULATION_SIZE)]

    ELITISM_SIZE = int(0.3 * POPULATION_SIZE)
    MAX_ITER = 30
    for i in range(MAX_ITER):
        population.sort()
        newPopulation[:ELITISM_SIZE] = population[:ELITISM_SIZE]
        for j in range(ELITISM_SIZE, POPULATION_SIZE, 2):
            parent1Index = selection(population)
            parent2Index = selection(population)

            crossover(population[parent1Index], population[parent2Index],
                newPopulation[j], newPopulation[j+1])

            mutation(newPopulation[j])
            mutation(newPopulation[j+1])

            newPopulation[j].fitness = newPopulation[j].calculateFitness()
            newPopulation[j + 1].fitness = newPopulation[j +
                1].calculateFitness()


        simulatedAnnealing(newPopulation[0], 10)
        population = newPopulation

    bestIndividual = max(population, key=lambda x: x.fitness)
    print('Solution: {}, fitness: {}'.format(bestIndividual.code,
        bestIndividual.fitness))
```

3.2.2 Fitnes funkcija

Ono što se javlja kao problem u oba navedena algoritma je što fitnes funkcija isključivo zavisi od preciznosti modela. Može da se desi da model koji ima malo bolju preciznost sadrži mnogo više atributa, ali je klasifikovan kao bolji. Zbog toga uvodimo i broj atributa u računanje fitnes funkcije. Koristeći `StandardScaler` možemo da kvantifikujemo ove vrednosti - jako male i jako velike vrednosti će dovoljno uticati na fitnes, tako da dajemo veću prednost jedinkama sa manjim brojem atributa. Sledi nova funkcija za izračunavanje fitnesa:

```
def calculateFitness(code, accuracy):
    scaler = StandardScaler()
```

```
    max_attributes = len(code)
    num_attributes = len([x for x in code if x])
    arr = np.array([1.0, num_attributes, max_attributes]).reshape(-1, 1)
    scaler.fit(arr)
    arr = scaler.transform(arr)
    [_, penalty, _] = scaler.fit_transform(X=arr)
    return accuracy - penalty
```

4 Rezultati

Uporedićemo rešenja za malo `n` koja daje genetski algoritam sa jednostavnim algoritmom grube sile. Algoritam za svako moguće rešenje izračunava preciznost učenja, što znači da je složenost $O(2^n)$, dok je složenost genetskog algoritma polinomijalna, i zavisi od postavljenih parametara (broja generacija i velicine populacije). U slučaju predstavljenih algoritama, eksperimentalnom metodom došlo se do `POPULATION_SIZE = 20` i `MAX_ITER = 20` vrednosti. Veličina populacije treba da bude dovoljno velika da generiše dovoljno raznovrsne jedinke. Za broj iteracija je utvrđeno da je najmanji koji daje tačan, odnosno dovoljno dobar rezultat. Prikazaćemo vreme izvršavanja algoritma za svako `n` , kao i vreme izvršavanja `calculateFitness` funkcije za različito `n`.
Sledi kod algoritma grube sile:

```
import itertools

def bruteForceAlg(n):
    numResources = n
    bestResult = ([True for x in range(numResources)], 0.0)
    numFeatures = n
    lst = [list(i) for i in itertools.product([False, True],
        repeat=numResources)]
    # skip first element [0,....,0]
    for i in range(1, len(lst)):
        sample = lst[i]
        print(sample)
        accuracy = getModelAccuracy(sample)
        print(accuracy)
        if accuracy > bestResult[1]:
            numFeaturesSample = len([x for x in sample if x])
            bestResult = (sample, accuracy)
            numFeatures = numFeaturesSample
        elif accuracy == bestResult[1]:
            numFeaturesSample = len([x for x in sample if x])
            if numFeaturesSample < numFeatures:
                bestResult = (sample, accuracy)
                numFeatures = numFeaturesSample

    print('Solution: {}, accuracy: {}', bestResult[0], bestResult[1])
```

n	BruteForce	GenAlg	GenAlgSimulatedAnnealing
4	[0, 0, 1, 1] 0.6505746841430664 25.2s	[0, 0, 1, 1] 0.6620689630508423 2min 42s	[0, 0, 1, 1] 0.659770131111145 7min 18s
5	[1, 0, 1, 1, 0] 0.657471239566803 42.8s	[1, 1, 0, 1, 1], 0.6666666865348816 2min 33s	[0, 0, 1, 1, 1], 0.6666666865348816 7min 29s
6	[1, 0, 1, 1, 0, 1] 0.6643677949905396 1min27s	[0, 0, 1, 1, 1, 0], 0.6666666865348816 2min 26s	[1, 0, 1, 1, 0, 1], 0.659770131111145 7min 27s
7	[0, 0, 1, 0, 1, 0, 1] 0.6666666865348816 3min4s	[0, 0, 1, 0, 1, 1, 0], 0.6643677949905396 2min 30s	[1, 1, 0, 1, 0, 0, 1], 0.6643677949905396 7min 25s
8	[1, 1, 1, 1, 0, 0, 0, 1] 0.6735632419586182 5min21s	[1, 0, 1, 1, 0, 0, 1, 0], 0.6712643504142761 2min 35s	[0, 1, 1, 1, 0, 0, 0, 1], 0.6689655184745789 7min 32s
9	[1, 0, 1, 1, 0, 1, 0, 0, 0] 0.6712643504142761 12min 15s	[1, 1, 1, 1, 0, 1, 1, 0, 0], 0.6689655184745789 2min 33s	[1, 0, 0, 1, 0, 1, 0, 1, 0], 0.6689655184745789 7min 24s
10	[1, 1, 0, 1, 0, 0, 0, 1, 0, 1] 0.6804597973823547 24min 1s	[1, 1, 0, 1, 0, 0, 0, 1, 0, 1], 0.6781609058380127 2min 40s	[0, 0, 1, 1, 0, 0, 0, 0, 0, 1], 0.6666666865348816 7min 33s

Tabela 1: rezultati algoritama nad malim `n`

4.1 Diskusija rezultata

Primetimo da se vreme izvršavanja algoritma grube sile značajno povećava za veće `n`. Zaključujemo da je algoritam nemoguće koristiti da veću dimenziju ulaza. Sa druge strane, genetski algoritam se izvršava u konstantom vremenu - menjanjem broja iteracija ili veličine populacije ovo vreme će varirati, ali vidimo da je algoritam koristan i za veliko `n`. Slično i za simulirano kaljenje, koje se duže izvršava zbog dodatnih komputacija, ali i dalje u konstantnom vremenu.
Treba uzeti u obzir da trenirani model za isti ulaz neće uvek vratiti isti izlaz. U slučajevima gde više podskupova ima jako slične performanse, ne znamo koji od njih će nam algoritam vratiti. Da bismo se osigurali da dobijemo najmanji mogući podskup, možemo da gledamo preciznost kao opseg, i fitnes računamo uzimajuci veličinu skupa u obzir. Predstavićemo i rezultate algoritma sa izmenjenim računanjem fitnesa.

n	GenAlg	GenAlg (expanded fitness)
5	[0, 0, 1, 0, 1] 0.6620689630508423 2min 58s	[0, 0, 1, 0, 0] 0.634482741355896 2min 54s
10	[1, 1, 1, 1, 0, 0, 0, 1, 0, 0], 0.6781609058380127 2min 40s	[1, 0, 0, 0, 0, 1, 0, 1, 0, 0], 0.6505746841430664 2min 45s
20	[0, 0, 0, 0, 1, 0, 0, 1, 0, 1, 1, 1, 0, 0, 1, 0, 0, 0, 0, 1] 0.7195402383804321 2min 49s	[0, 0, 1, 0, 1, 1, 0, 1, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0] 0.707011517683665 2min 57s

Tabela 2: rezultati genetskog algoritma sa proširenim računanjem fitnesa

Vidimo da u nekim slučajevima možemo da dobijemo dosta manje skupove atributa koji imaju sličnu preciznost. Na primeru `n=20` rezultat običnog i proširenog algoritma je isti (ista veličina skupa), pa možemo zaključiti da je podksup veličine 7 sigurno dobro rešenje.

n	model	model + attribute set size
5	2.11s	1.72s
10	1.71s	1.72s
20	1.73s	1.75s

Tabela 3: vreme izvršavanja računanja fitnesa samo sa modelom, i sa modelom + brojem atributa za različito `n`

5 Zaključak

Ukoliko analiziramo same atribute, ne treba gledati vrednost pojedinačnih atributa, već veze između njih, odosno vrednost kombinacija atributa.
Mana predloženog algoritma je u ceni računanja fitnesa jedinke. Za svaku novu jedinku moramo da treniramo mašinu sa novim podskupom atributa, da bismo ocenili tačnost klasifikacije nad tim podskupom. Vreme trajanja ove operacije nije zanemarljivo, i bilo bi bolje na drugi način oceniti fitnes.
Sa druge strane, genetski algoritam u razumljivo vreme daje rezultat koji može da parira priloženim rezultatima pohlepnog algoritma. Jedna varijacija algoritma bi mogla biti da ocena fitnesa jedinke bude kombinacija tačnosti klasifikacije i broja atributa, gde se prednost daje jedinkama sa manjim brojem atributa.

Literatura

[1] K. Van Horn and T. Martinez, *The Minimum Feature Set Problem*, Neural Networks 7 (1994), no. 3, pp. 491-494. https://axon.cs.byu.edu/papers/vanhorn_3.pdf.

[2] K.Kira, L.Randell, *The Feature Selection Problem: Traditional Methods and a New Algorithm*, AAAI-92 Proc., 10th International Conference on Artificial Intelligence, 1992.

[3] M.Cui, S.Prasad, M.Mahrooghy, *Genetic algorithms and Linear Discriminant Analysis based dimensionality reduction for remotely sensed image analysis*, 2011 IEEE International Geoscience and Remote Sensing Symposium, available at https://www.researchgate.net/publication/220819539_Genetic_algorithms_and_Linear_Discriminant_Analysis_based_dimensionality_reduction_for_remotely_sensed_image_analysis.