Solving the feature set problem with Genetic Programming

Anja Miletic

August 29, 2021

Contents

1 Uvod

Problem biranja najmanjeg podskupa atributa (eng. Minimum feature subset selection problem) je bitan u istrazivanju podataka i masinskom ucenju. Kolicina podataka je sve veca, pa su tehnike ciscenja, odnosno smanjivanja obimnosti podataka vaznije. U slucaju najmanjeg podskupa atributa, cilj je smanjiti dimenzionalnost podataka bez gubljenja tacnosti klasifikacije, radi brzeg treniranja neuronskih mreza.

2 Metode

Dati problem je NP tezak, i tesko ga je aproksimirati [1]. Postoje razliciti pristupi resavanju problema. Nabrojimo neke [2]:

- selekcija pojedinacnih atributa koristeci metrike koje se odnose na vaznost tih atributa. Naravno ova metoda je brza i deluje jednostavno, ali ne uzima u obzir odnos izmedju atributa, samim tim ne daje najbolje rezultate.
- potpuna pretraga prostora resenja. Od svih valjanih podskupova atributa izabrati onaj koji maksimizuje tacnost. Ova metoda je optimalna - od svih mogucih resenja ona ce dati najmanji podskup. Zbog velikog broja mogucih resenja, potpuna pretraga se moze koristiti samo ako je ukupan broj atributa, odnosno ukupan broj mogucih podskupova mali
- heuristicke 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 tezinama atributa, algoritam prepoznaje one atribute koji su statisticki bitni, proucavajuci razlike izmedju vrednosti podataka. Algoritam je polinomijalne slozenosti.

3 Predlog resenja koriscenjem genetskog algoritma

Genetski algoritmi pripadaju porodici optimizacionih algoritama koji traze globalni minimum fitnes funkcije. U opstem slucaju, to ukljucuje cetiri koraka: [3]

- evaluacija: bira se pseudo-slucajan set pocetne populacije. Izracunava se fitnes svake jedinke, zatim se one sortiraju na osnovu fitnesa.
- reprodukcija: biraju se najbolje jedinke koje se cuvaju u narednoj generaciji. Ove jedinke se zovu elitna deca.
- rekombinacija: biraju se jedinke koje ce se ukrstiti da bi se napravile jedinke za sledecu generaciju. U ovom koraku pokusavamo da sacuvamo najbolje gene i kombinujemo ih da napravimo jos 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 resenja

Koristicemo genetski algoritam da nadjemo najmanji skup atributa sa najvecom preciznoscu. Jedinke ce biti kodirane binarnim nizom duzine m, gde je m ukupan broj atributa naseg 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 ucen nad podacima koji sadrze podskup atributa predstavljen kodom jedinke. Koristicemo sekvencijalni model `keras` biblioteke koji je optimizovan za probleme klasifikacije.

3.1.1 Podaci

Koristicemo podatke sa kosarkaskih utakmica NBA sezone 2020-21. Atributi se odnose na razne statisticke parametre koji se prate tokom utakmice, a zabelezeni 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 ciscenja podataka ostaje nam 21 numericki atribut, kao i kolona `W\\L` koju mapiramo kao `W->1, L->0`.

3.1.2 Algoritam

U nastavku sledi kod za odredjivanje 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`, tacnost klasifikacije validacionog skupa, da bismo izbegli slucajeve gde je `accuracy = 100.0` zbog preprilagodjavanja.
Klasa `Individual` sadrzi 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 slucaju biramo nasumicno jedan atribut koji ce pripadati skupu.
Implementiracemo dve verzije genetskog algoritma - obican i algoritam sa simuliranim kaljenjem. Simulirano kaljenje je jos jedna tehnika koja omogucuje izlazak iz lokalnog minimuma, tako sto se prihvata gore resenje, sa verovatnocom obrnuto-proporcionalnom broju iteracija.
Pre njih, uvodimo pomocne funkcije selekcije, ukrstanja 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()
```

Koristicemo turnirsku selekciju i jednopoziciono ukrstanje. 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))


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))
```

4 Rezultati

Uporedicemo resenja za malo n koja daje genetski algoritam sa jednostavnim algoritmom grube sile. Algoritam za svako moguce resenje izracunava preciznost ucenja, sto znaci da je slozenost $O(2^n)$, dok je slozenost genetskog algoritma polinomijalna, i zavisi od postavljenih parametara (broja generacija i velicine populacije).

Treba uzeti u obzir da trenirani model za isti ulaz nece uvek vratiti isti izlaz. U slucajevima gde vise podskupova ima jako slicne performanse, ne znamo koji od njih ce nam algoritam vratiti. Da bismo se osigurali da dobijemo najmanji moguci podskup, mozemo da gledamo preciznost kao opseg, i fitnes racunamo uzimajuci velicinu skupa u obzir.

n	BruteForce	GenAlg	GenAlgSimulatedAnnealing
4	[0, 0, 1, 1] 0.6505746841430664	[0, 0, 1, 1] 0.6620689630508423	[0, 0, 1, 1] 0.659770131111145
5	[1, 0, 1, 1, 0] 0.657471239566803	[1, 1, 0, 1, 1], 0.6666666865348816	[0, 0, 1, 1, 1], 0.6666666865348816
6	[1, 0, 1, 1, 0, 1] 0.6643677949905396	[0, 0, 1, 1, 1, 0], 0.6666666865348816	[1, 0, 1, 1, 0, 1], 0.659770131111145
7	[0, 0, 1, 0, 1, 0, 1] 0.6666666865348816	[0, 0, 1, 0, 1, 1, 0], 0.6643677949905396	[1, 1, 0, 1, 0, 0, 1], 0.6643677949905396
8	[1, 1, 1, 1, 0, 0, 0, 1] 0.6735632419586182	[1, 0, 1, 1, 0, 0, 1, 0], 0.6712643504142761	[0, 1, 1, 1, 0, 0, 0, 1], 0.6689655184745789
9	[[1, 0, 1, 1, 0, 1, 0, 0, 0] 0.6712643504142761	[1, 1, 1, 1, 0, 1, 1, 0, 0], 0.6689655184745789	[1, 0, 0, 1, 0, 1, 0, 1, 0], 0.6689655184745789
10	[1, 1, 0, 1, 0, 0, 0, 1, 0, 1] 0.6804597973823547	[1, 1, 0, 1, 0, 0, 0, 1, 0, 1], 0.6781609058380127	[0, 0, 1, 1, 0, 0, 0, 0, 0, 1], 0.6666666865348816

Table 1: rezultati algoritama nad malim `n`

5 Zakljucak

Ukoliko analiziramo same atribute, ne treba gledati vrednost pojedinacnih atributa, vec veze izmedju njih, odosno vrednost kombinacija atributa.
Mana predlozenog algoritma je u ceni racunanja fitnesa jedinke. Za svaku novu jedinku moramo da treniramo masinu sa novim podskupom atributa, da bismo ocenili tacnost klasifikacije nad tim podskupom. Vreme trajanja ove operacije nije zanemarljivo, i bilo bi bolje na drugi nacin oceniti fitnes.
Sad druge strane, genetski algoritam u razumljivo vreme daje rezultat koji moze da parira prilozenim rezultatima pohlepnog algoritma. Jedna varijacija algoritma bi mogla biti da ocena fitnesa jedinke bude kombinacija tacnosti klasifikacije i broja atributa, gde se prednost daje jedinkama sa manjim brojem atributa.

References

[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.