DD1418 Språkteknologi med introduktion till maskininlärning
Johan Boye
2020-11-13

Uppgift 4

Litteratur: Läs kapitel 5 i Jurafsky-Martin, samt titta på föreläsning 8.

Kod: Skelettkoden kan laddas ned från Canvas eller från

`http://www.csc.kth.se/~jboye/teaching/language_engineering/a04/NER.zip`

Unzippa koden i lämplig mapp. Öppna ett kommandofönster, gå till foldern `HMM` och skriv:

```
pip install -r requirements.txt
```

Nu ska allting du behöver för att göra labben vara installerat.

Problem:
I denna uppgift kommer vi att undersöka användningen av binär logistisk regression i så kallad "named entity recognition", dvs att hitta ord i en text som är en del av ett namn. Din huvuduppgift är att **utöka programmet** `BinaryLogisticRegression.py` **för att få det att träna en binär logistisk regressionsmodell från en träningsmängd, och att använda modellen för att klassifiera ord från en testmängd som antingen 'namn' eller 'inte namn'**.

Öppna träningsfilen `ner_training.csv`. Varje rad innehåller ett ord och en etikett. Om etiketten är 'O' innebär det att ordet inte är ett namn; annars är ordet ett namn av någon sort. För tillfället kommer vi att betrakta alla olika sorters namn som "namn".

Klassen `NER.py` läser in en korpus på detta format och transformerar den till en vektor med etiketter och en vektor med särdragsvärden. Etiketterna är antingen 1 (om ordet är ett namn), eller 0 (om det inte är ett namn). Det finns två särdrag: det första särdraget har värdet 1 om ordet börjar med stor bokstav, och 0 om det inte gör det. Det andra särdraget har värdet 1 om ordet är det första ordet i en mening, och 0 om det inte är det. Till exempel, från raden

```
Demonstrators,O
```

får vi etikett 0 eftersom ordet inte är ett namn, och särdragsvektorn $(1, 1)$ eftersom ordet börjar med en stor bokstav och är det första i en mening. Särdragen beräknas i metoderna `capitalized_token` och `first_token_in_sentence`.

Notera att när du kallar på klassen `BinaryLogisticRegression.py` adderas ett extra "dummy"-särdrag (som alltid är 1) till varje datapunkt. Datapunkterna är därför representerade som en matris `x` av storlek `DATAPOINT` × (`FEATURES` + `1`) och korresponderande kategori som en vektor `y` av längd `DATAPOINT`.

1. Utöka klassen `BinaryLogisticRegression.py`: metoden `fit` ska implementera *batch gradient descent* för att beräkna modellparametern θ (som är en vektor), där θ_0 är bias-vikten och θ_1 och θ_2 är vikterna för respektive särdrag 1 och 2. Metoden `conditionalProb` ska beräkna den betingade sannolikheten $P(etikett|d)$, där *etikett* är antingen 1 eller 0, och d är indexet för datapunkten. Testa din modell på testmängden `ner_test.csv` genom att köra skriptet

`run_batch_gradient_descent.sh`. För att se algoritmens framsteg kan du plotta gradienten (se problem 2 nedan).

Batch gradient descent: m är antalet datapunkter, n är antalet särdrag, α är inlärningshastigheten ("learning rate"). Konvergens uppnås när summan av kvadraterna av de partiella derivatorna `gradient[k]` är mindre än konstanten `CONVERGENCE_MARGIN`.

```
Repetera tills konvergens:
   för k = 0 till n:
      gradient[k] = 1/m Σ_{i=1}^{m} x_k^(i) (σ(θ^T x^(i)) − y^(i))
   för k = 0 till n:
      θ[k] = θ[k] − α× gradient[k]
```

2. Följ hur algoritmen konvergerar genom att stoppa in raden

   ```
   self.update_plot(self.loss(self.x, self.y))
   ```

   på ett lämpligt ställe i loopen (men var medveten om att programmet exekverar betydligt långsammare om man plottar varje iteration). *Notera att detta kommer att fungera bara efter att du har implementerat* `self.loss` *funktionen.* Om inlärningen är för långsam, testa att öka inlärningshastigheten. OBS. att du ska bara räkna ut och plotta värdet på loss-funktionen tills du är säker att koden är korrekt (annars kommer träningen att ta för lång tid).

3. Lägg till kod i metoden `stochastic_fit` så att den implementerar *stochastic gradient descent* för att beräkna θ. Använd plottning för att se hur algoritmen konvergerar. Testa din kod genom att köra skriptet `run_stochastic_gradient_descent.sh`. Vad är skillnaden i prestanda i jämförelse med batch gradient descent?

```
Repetera ett fixt antal gånger (t.ex. 10m), eller tills konvergens:
     Välj i slumpmässigt, 0 ≤ i ≤ m:
     för k = 0 till n:
          gradient[k] = x_k^(i) (σ(θ^T x^(i)) − y^(i))
     för k = 0 till n:
          θ[k] = θ[k] − α×gradient[k]
```

4. Lägg till kod till metoden `minibatch_fit` så att den implementerar minibatch gradient descent. Använd plottning för att se hur algoritmen konvergerar. Testa din kod genom att köra `run_minibatch_gradient_descent.sh`. Vad blir skillnaden i prestanda jämfört med föregående varianter av gradient descent?

5. Beräkna modellens **accuracy** givet testmängden, **precision** och **recall** för etiketterna "name" och "no name". Presentera dina resultat och förklara hur du beräknade dem.

6. Försök förbättra resultaten genom att lägga till ett nytt särdrag, ändra ett befintligt särdrag och/eller genom regularisering (regularization).