UNIVERSITÀ DEGLI STUDI FIRENZE

Data Mining

Rossman Store Sales competition: addestramento di un modello di DM per la previsione di vendite

Tommaso Papini
tommaso.papini1@stud.unifi.it
5537529

In questo breve elaborato verranno descritte l'implementazione ed il funzionamento dello script python che il sottoscritto ha realizzato per la partecipazione alla competizione Kaggle *Rossman Store Sales*[1] e come progetto per il corso di Data Mining (a.a. 2015/2016).

Il codice aggiornato dello script può essere trovato al seguente indirizzo: `https://github.com/oddlord/rossman-store-sales`.

Modello e feature

Il funzionamento dello script è variato molto dalla sua prima versione. Vediamo brevemente quali sono stati i principali cambiamenti.

Nelle prime versioni si è provato a fare una **media** globale e quindi una media per store delle vendite in fase di training, per poi usare tali medie come predizione in fase di testing. In questi casi il valore dell'errore *rmspe* era attorno a 0.25.

Successivamente si è provato ad utilizzare il pacchetto *sklearn*, che fornisce molti strumenti per il Data Mining.
La prima prova con *sklearn* è stata di utilizzare un *LinearRegressor* come modello, passandogli un po' tutti i campi del trainset come feature. In particolare si scelsero come feature *Store*, *DayOfWeek*, *Promo*, *StateHoliday* e *SchoolHoliday*. Veniva utilizzato anche il campo *Open* ma non come feature vera e propria, ma per settare direttamente la predizione a 0 nel caso in cui lo store fosse chiuso (store chiusi in fase di training non venivano considerati). Questo primo approccio con *sklearn* dette risultati molto bassi (intorno a 0.45), addirittura peggiori delle medie fatte inizialmente.
Si è quindi cambiato il modello con un altro regressore, ovvero il **KNeighborsRegressor**, utilizzando le

[1] `https://www.kaggle.com/c/rossmann-store-sales`

stesse feature di prima: in questo caso i risultati sono notevolmente migliorati, ottenendo un *rmspe* di circa 0.149.
Infine, sempre con le stesse feature, si è cambiato il modello in un **DecisionTreeRegressor**, ottenendo score leggermente più bassi ma sempre dell'ordine di 0.149.

Un miglioramento notevole, da 0.149 a 0.14066 si è avuto quando si sono introdotti i modelli **per-store**, ovvero assegnando (e addestrando) un modello diverso per ogni store, ognuno dedicato alle istanze relative a quello store. Non solo, questo miglioramento è anche dovuto al fatto di aver notevolmente diminuito il numero di feature utilizzate: mentre prima si usavano un po' tutti i campi del trainset, in questa fase si sono invece passati soltanto **DayOfWeek** e **Promo**. Per prove ed errori si è stabilito che le feature scartate aumentavano l'errore totale anziché diminuirlo. Da questo se ne è dedotto che tra i campi presenti nel trainset, gli unici veramente significativi sono quei due (escludendo *Store* e *Open*, che da qui in avanti non saranno più considerati vere e proprie feature).

Si è provato, a questo punto, ad utilizzare metodi ensemble, quali *Bagging*, *Boosting* e *Random Forests*. I risultati sono tuttavia stati piuttosto insoddisfacenti: i tempi d'esecuzione erano notevolmente allungati, in alcuni casi risultavano circa il doppio o il triplo rispetto agli alberi di decisione, mentre gli errori risultavano o uguali o leggermente più alti. A questo si aggiungeva la componente casuale dei metodi ensemble, che rende più difficile ricreare più volte certi risultati. Per questi motivi i metodi ensemble sono stati scartati, scegliendo di rimanere sui Decision Trees.
Dopodiché si sono cercati altri metodi, al di fuori dei campi del trainset, per migliorare ulteriormente il risultato.

Un'idea è stata quella di usare come feature i campi *StateHoliday* e *SchoolHoliday* del giorno seguente. Intuitivamente, un negozio non venderà di più il giorno di Natale (in cui sarà probabilmente chiuso), ma il giorno prima! Per tentare di fare questo in modo efficiente, ovvero evitando di scorrere ogni volta il dataset, si era pensato di scrorrere una prima volta i dataset, andando a salvare in un dizionario le informarioni relative a ogni giornata. A questo proposito, venivano estratte le informazioni soltanto dalle istanze di ogni data relative allo store con id 1, dopotutto se un giorno è festa lo è indipendentemente dal negozio... no? Sbagliato! Quest'affermazione è falsa in quanto non considera la diversa localizzazione dei negozi (purtroppo anonima) e del fatto che in posti diversi potrebbero essere festeggiate feste diverse. Si osservi ad esempio la data 15 Agosto 2015 nel testset: per alcuni negozi è riportata come festa, mentre per altri come giorno normale! Facendo una piccola ricerca[2] si trova subito che in Germania (Rossman è una catena tedesca) il 15 Agosto viene festeggiata l'Assunzione di Maria Vergine, ma sono nelle regioni del Saarland e della Baviera! Quest'approccio è risultato quindi sbagliato e l'idea è stata accantonata.

Da un *rmspe* di 0.14066 si è riusciti a scendere a 0.13476 aggiungendo un'ulteriore feature, relativa alla distanza dal negozio più vicino che rappresenta un qualche tipo di **concorrenza**. L'idea è semplice: per ogni store, estrai il campo *CompetitionDistance* dal file *store.csv* e usa la distanza come feature solo nelle date successive a quella indicata. Se nessuna data è indicata, si assume che il negozio in questione era lì già da prima che venissero raccolti dati a riguardo, quindi si considera sempre presente dall'inizio del training. Se nessuna distanza è fornita, o per le date precedenti a quelle indicate, si assume che non ci sia uno store abbastanza vicino da essere considerato in concorrenza: in questo caso si assegna un valore infinito (*sys.maxint*) alla distanza.

Visti i buoni risultati ottenuti con *CompetitionDistance* si è quindi provata una cosa simile per *Promo2*. Leggendo la descrizione e andando a documentarsi nel forum della competizione si capisce che se un negozio lancia la Promo2, allora la lancia a partire da una certa settimana di un certo anno e ogni tre mesi la rinnova (*PromoInterval*). Sul forum gli admin hanno spiegato che *Promo2* rappresenta delle promozioni basate su coupon di sconto che valgono 3 mesi. Quindi, per gli store/date in cui Promo2 era attiva si è provato ad

[2] I giorni festivi in Germania: `http://www.viaggio-in-germania.de/festivi.html`

usare come feature la distanza (in giorni) tra la data attuale e l'inizio della mandata di coupon più recente. Tristemente l'errore è schizzato a 0.18 circa, rendendo inutile questa feature.

Si è quindi provato ad utilizzare la feature **Open** ma, stavolta, relativa al giorno successivo all'istanza corrente (per uno stesso negozio ovviamente). L'idea che giustifica questa prova sta nel fatto che solitamente le persone tendono a comprare di più il giorno prima che un negozio sia chiuso. Infatti non ha molto senso andare ad utilizzare come feature le flag di *StateHoliday* o *SchoolHoliday*, in quanto, ad esempio, il giorno di Natale i negozi saranno chiusi e l'istanza risulterà irrilevante. Ciò che può essere interessante è invece il giorno della Vigilia di Natale, in cui solitamente si registrano più incassi. Adottando questa feature si è migliorato ulteriormente l'errore scendendo a 0.13338.

Si è quindi cercato individuare in qualche modo gli **outlier**, ovvero i valori che più si discostano del trainset, in modo da poterli escludere dall'addestramento per rendere il processo più stabile. Un'idea molto semplice è stata di calcolare la media di vendite per ogni negozio e quindi definire come outlier tutte le istanze che si allontanavano più di un tot dalla media calcolata. Si sono fatte varie prove per cercare il valore migliore e si è trovato che considerare le istanze che si allontanano più del 250% della media da i risultati migliori. Tuttavia i miglioramenti non sono stati grandiosi, in quanto si è migliorato di soltanto uno 0.00002.

Dopodiché si è tentato di riutilizzare alcune feature scartate in precedenza (come il mese o l'anno) scomponendole in una lista di feature binarie, ovvero enumerando i possibili valori che tale feature può assumere e creando una feature binaria per ogni valore. Purtroppo i risultati sono stati comunque negativi. Si è provato anche rendendo binaria la feature *DayOfWeek*, ma questo fa aumentare l'errore rispetto alla feature originaria.

Un'ultima feature provata è la media delle vendite durante la settimana precedente all'istanza analizzata (intendendo come settimana non i sette giorni precedenti ma dal lunedì alla domenica). Per implementare questa feature si è creato un dizionario all'inizio dell'esecuzione contenente tutte le medie di tutte le settimane (del periodo di training) per tutti i negozi. Per le settimane di test invece le medie venivano calcolate e aggiornate via via che venivano fatte le predizioni, utilizzando il valore predetto per calcolare la media (assumendo un certo accumulo di errore ovviamente). I risultati sono tuttavia stati deludenti, con l'errore che è salito a circa 0.19.

Implementazione

Il codice dello script è suddiviso in quattro file python:

- pred.py
- features.py
- utils.py
- config.py

Il file di configurazione **config.py** serve semplicemente per permettere all'utente di specificare in modo rapido i percorsi dei vari dataset utilizzati per l'addestramento e la predizione, senza dover utilizzare ogni volta opzioni da linea di comando.
Il file **utils.py** raccoglie tutte le funzioni di svariata utilità, come le funzioni per fare il parsing delle opzioni da linea di comando (e controllarne la validità) o per il calcolo dell'errore *RMSPE* o per il logging del tempo di esecuzione delle varie attività dello script.
features.py raccoglie tutte le funzioni necessarie ad estrarre i vari campi dai vari dataset. In particolare viene definito un grande dizionario *fields* dove si definisce, per ogni campo, in che posizione si trova nei vari

dataset (dato che stessi campi possono avere posizioni diverse all'interno di dataset diversi) e con che funzione eseguire il parsing di tale campo (infatti noi andiamo a leggere una stringa dal dataset ma in generale vogliamo ottenere un valore numerico).
Infine il file **pred.py** è il file principale dello script. In questo file viene eseguita innanzitutto una fase di estrazione delle feature (a seconda delle feature scelte, si veda la sezione precedente), seguita da una fase di addestramento e quindi una fase finale di predizione.

Per lanciare lo script basta eseguire il file *pred.py* (specificando opportunamente l'interprete python), seguito dalle eventuali opzioni. Per una descrizione dettagliata delle opzioni basta eseguire *pred.py -h* ma a grandi linee esse permettono di specificare file alternativi a quelli indicati in *config.py*, di attivare la compressione dei risultati o di attivare la fase di validazione (eventualmente visualizzando un grafico).

Nella fase di **estrazione delle feature** vengono chiamate delle funzioni apposite per estrarre le feature delle istanze dei dataset di train e di test (eventualmente di validation) e restituirle sotto forma di liste X e Y (per train e validation) o soltanto X (per il testset). Tutte e tre le funzioni che creano le liste sopracitate richiamano, al loro interno, una funzione *extract_instance* che si occupa di estrarre le feature di una particolare istanza (di train o di test) sotto forma di lista di valori numerici. Questo è fatto per ragioni di comodità e consistenza: in questo modo saremo sempre sicuri di andare a estrarre le stesse feature (e posizionarle nello stesso ordine) per i vari dataset. Si deve precisare il fatto che, nel caso del trainset, la X e la Y non sono esattamente liste, ma piuttosto dizionari di liste indicizzati secondo lo store id: in questo modo si tengono separate le istanze dei vari store per permettere il training suddiviso per store (ogni elemento del dizionario sarà una lista di istanze relative ad un particolare store). Per ragioni analoghe, negli elementi delle liste per i dataset di validation e di test, si inseriscono dati a contorno delle feature, in modo, ad esempio, da poter recuperare immediatamente lo store id corrispondente o lo stato di chiusura di ogni istanza.

Successivamente, durante la fase di **training** si scorrono tutti gli indici degli store e per ognuno di essi si crea un nuovo modello e lo si addestra (*fit*) passandogli le corrispondenti liste di input e di target.

Dopo il training si ha la fase, opzionale, di **validation** in cui si prendono gli ultimi due mesi del trainset (se la validation è attiva questi due mesi sono esclusi dal training!) e si calcola il valore *rmspe* sui valori predetti dai vari modelli su questi dati. Dopo si può anche scegliere di plottare un semplice grafico relativo alle date di validation per capire dove/come/perché si era sbagliato.

Infine si ha la fase di **predizione** in cui, in modo del tutto analogo a quanto visto per la fase di validation, si calcolano i valori predetti di vendite per i vari store e per ognuno di questi si inserisce una voce (indicizzata univocamente con l'id dell'istanza) in un nuovo file, chiamato di default *predictions.csv*. Se l'utente lo desidera, si può decidere di comprimere tale file utilizzando la tecnica GZip.