Auto-codificatore LSTM per la classificazione di eventi rari estremi in telecamere

Qui apprenderemo i dettagli della preparazione dei dati per i modelli LSTM e costruiremo un codificatore automatico LSTM per la classificazione di eventi rari.

Chitta Ranjan
18 maggio · 13 minuti di lettura

Questo post è una continuazione del mio precedente post Extreme Rare Event Classification che utilizza Autoencoder . Nel post precedente, abbiamo parlato delle sfide in un evento estremamente raro con dati con meno dell'1% di dati con etichetta positiva. Abbiamo creato un classificatore di autoencoder per tali processi utilizzando i concetti di Anomaly Detection.

Tuttavia, i dati che abbiamo sono una serie temporale. Ma in precedenza abbiamo usato un Autoencoder a livello denso che non utilizza le funzionalità temporali nei dati. Pertanto, in questo post, miglioreremo il nostro approccio creando un Autoencoder LSTM.

Qui impareremo:

- fasi di preparazione dei dati per un modello LSTM,
- costruire e implementare il codificatore automatico LSTM e
- utilizzando il codificatore automatico LSTM per la classificazione di eventi rari.

Riepilogo rapido su LSTM :

- LSTM è un tipo di rete neurale ricorrente (RNN). RNN, in generale, e LSTM, in

particolare, vengono utilizzati su dati sequenziali o di serie temporali.

- Questi modelli sono in grado di estrarre automaticamente l'effetto di eventi passati.
- LSTM è noto per la sua capacità di estrarre effetti sia a lungo che a breve termine dell'evento del passato.

Di seguito, andremo direttamente allo sviluppo di un codificatore automatico LSTM. Si consiglia di leggere la comprensione passo-passo dei livelli di Lencm Autoencoder per comprendere meglio e migliorare ulteriormente la rete sottostante.

In breve, in merito al problema dei dati, disponiamo di dati reali sulle interruzioni dei fogli di una produzione di carta. Il nostro obiettivo è prevedere la pausa in anticipo. Per i dettagli su dati, problemi e approccio alla classificazione, fare riferimento alla classificazione degli eventi rari estremi utilizzando gli autocodificatori .

Auto-codificatore LSTM per dati multivariati

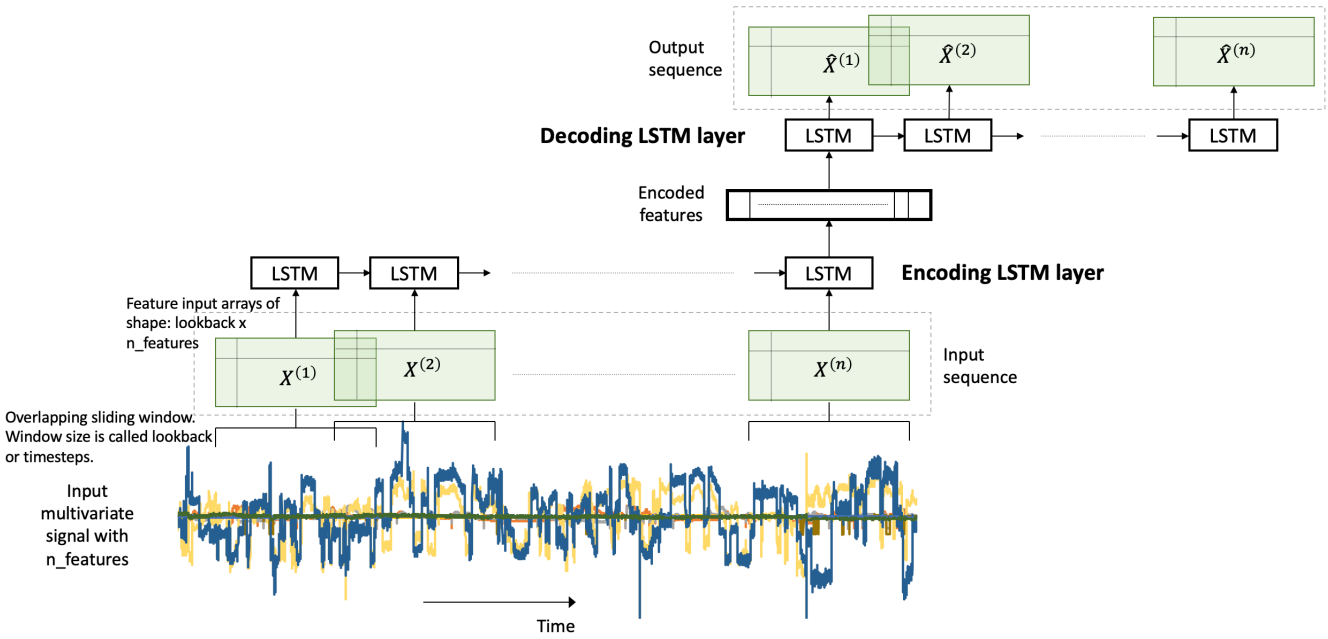


Figura 1. Un codificatore automatico LSTM.

Nel nostro problema, disponiamo di dati multivariati relativi alle serie temporali. I dati di una serie temporale multivariata contengono più variabili osservate per un periodo di tempo. Costruiremo un autoencoder LSTM su questa serie temporale multivariata per eseguire la classificazione di eventi rari. Come descritto in [1], ciò si ottiene utilizzando un approccio di rilevazione delle anomalie:

- costruiamo un codificatore automatico sui dati *normali* (con etichetta negativa),

- usalo per ricostruire un nuovo campione,
- se l'errore di ricostruzione è elevato, lo etichettiamo come una rottura del foglio.

LSTM richiede alcuni passaggi speciali per la preelaborazione dei dati. Di seguito, presteremo sufficiente attenzione a questi passaggi.

Andiamo all'implementazione.

biblioteche

Mi piace prima mettere insieme le biblioteche e le costanti globali.

```
% matplotlib inline
import  matplotlib.pyplot  as  plt
import  seaborn  as  sns

import  pandas  as  pd
import  numpy  as  np
from  pylab  import rcParams

import  tensorflow  as  tf
from  keras  import optimizer, Sequential
from  keras.models  import Model
from  keras.utils  import plot_model
from  keras. i layer  importano Dense, LSTM, RepeatVector,
TimeDistributed
da keras.callbacks  importare ModelCheckpoint, TensorBoard

da  sklearn.preprocessing  importazione StandardScaler
da  sklearn.model_selection  importazione train_test_split
da  sklearn.metrics  importare confusion_matrix,
precision_recall_curve
da  sklearn.metrics  importare recall_score, classification_report,
AUC, roc_curve
da  sklearn.metrics  importare precision_recall_fscore_support,
f1_score

da  NumPy.  seme di importazione casuale
 seme (7)
da  tensorflow  import set_random_seed
set_random_seed (11)

da  sklearn.model_selection  import train_test_split

SEED = 123 #utilizzato per aiutare a selezionare casualmente i punti
dati
 DATA_SPLIT_PCT = 0.2

rcParams ['figure.figsize'] = 8, 6
LABELS = ["Normal", "Break"]
```

Preparazione dei dati

Come accennato in precedenza, LSTM richiede alcuni passaggi specifici nella preparazione dei dati. L'input per gli LSTM sono array tridimensionali creati dai dati delle serie temporali. Questo è un **passo soggetto a errori,** quindi esamineremo i dettagli.

Leggi i dati

I dati sono presi da [2]. Il link ai dati è qui .

```
df = pd.read_csv ("data / processminer-rare-event-mts - data.csv")
df.head (n = 5)   # visualizza i dati.
```

	time	y	x1	x2	x3	x4	x5	x6	x7	x8	...	x52	x53	x54
0	5/1/99 0:00	0	0.376665	-4.596435	-4.095756	13.497687	-0.118830	-20.669883	0.000732	-0.061114	...	10.091721	0.053279	-4.936434
1	5/1/99 0:02	0	0.475720	-4.542502	-4.018359	16.230659	-0.128733	-18.758079	0.000732	-0.061114	...	10.095871	0.062801	-4.937179
2	5/1/99 0:04	0	0.363848	-4.681394	-4.353147	14.127998	-0.138636	-17.836632	0.010803	-0.061114	...	10.100265	0.072322	-4.937924
3	5/1/99 0:06	0	0.301590	-4.758934	-4.023612	13.161567	-0.148142	-18.517601	0.002075	-0.061114	...	10.104660	0.081600	-4.938669
4	5/1/99 0:08	0	0.265578	-4.749928	-4.333150	15.267340	-0.155314	-17.505913	0.000732	-0.061114	...	10.109054	0.091121	-4.939414

5 rows × 63 columns

Cambio di curva

Come menzionato anche in [1], l'obiettivo di questo problema di eventi rari è prevedere una rottura del foglio prima che si verifichi. Proveremo a prevedere l'interruzione **fino a** 4 minuti in anticipo. Per questi dati, ciò equivale a spostare le etichette in alto di due righe. Può essere fatto direttamente con `df.y=df.y.shift(-2)`. Tuttavia, qui è necessario eseguire le seguenti operazioni,

- Per ogni riga *n* con etichetta 1, crea (*n* -2) :(*n* -1) come 1. Con questo, stiamo insegnando al classificatore a prevedere **fino a** 4 minuti in anticipo. E,
- rimuovi riga *n* . La riga *n* viene rimossa perché non ci interessa insegnare al classificatore a prevedere un'interruzione quando è già avvenuta.

Sviluppiamo la seguente funzione per eseguire questo spostamento della curva.

```
sign = lambda x: (1, -1) [x <0]

def curve_shift (df, shift_by):
    ' ' '
    Questa funzione sposta le etichette binarie in un frame di dati.
    Lo spostamento della curva sarà rispetto a 1s.
    Ad esempio, se shift è -2,
    si verificherà il seguente processo : se la riga n è etichettata
come 1, allora
    - Crea riga (n + shift_by) :( n + shift_by-1) = 1.
    - Rimuovi riga n.
    cioè le etichette verranno spostate fino a 2 righe in alto.
Input:     df Un frame di dati panda con una colonna con etichetta
binaria.              Questa colonna etichettata dovrebbe essere
denominata 'y'.     shift_by Un numero intero che indica il numero
di righe da spostare.     Produzione




    df Un frame di dati con le etichette binarie spostate di shift.
    ' ' '

     vector = df [' y ']. copy ()
    per s nel range (abs (shift_by)):
        tmp = vector.shift (segno (shift_by))
        tmp = tmp.fillna (0)
        vector + = tmp
    labelcol = 'y'
    # Aggiungi vettore a df
     df.insert (loc = 0, colonna = labelcol + 'tmp', valore =
vettore)
    # Rimuovi le righe con labelcol == 1.
     df = df.drop (df [df [labelcol ] == 1] .index)
    # Rilascia labelcol e rinomina il tmp col come labelcol
     df = df.drop (labelcol, axis = 1)
    df = df.rename (colonne = {labelcol + 'tmp': labelcol})
    # Crea il etichetta binaria
    df.loc [df [labelcol]> 0, labelcol] = 1

    restituisce df
```

Sposteremo ora i nostri dati e verificheremo se lo spostamento è corretto. Nelle sezioni successive, abbiamo alcuni altri passaggi del test. Si consiglia di utilizzarli per assicurarsi che le fasi di preparazione dei dati funzionino come previsto.

```
print ('Prima dello spostamento')   # Righe etichettate positive
```

```
# prima dello spostamento.
one_indexes = df.index [df ['y'] == 1]
display (df.iloc [(np.where (np.array (input_y) == 1) [0] [0] -5) :(
np. dove (np.array (input_y) == 1) [0] [0] +1),])

# Sposta la colonna di risposta y di 2 righe per fare una previsione
in anticipo di 4 minuti.
df = curve_shift (df, shift_by = -2)

print ('After shifting')   # Convalida se lo spostamento è avvenuto
correttamente.
display (df.iloc [(one_indexes [0] -4) :( one_indexes [0] +1), 0: 5]
.head (n = 5))
```

```
Before shifting
```

	time	y	x1	x2	x3
256	5/1/99 8:32	0	1.016235	-4.058394	-1.097158
257	5/1/99 8:34	0	1.005602	-3.876199	-1.074373
258	5/1/99 8:36	0	0.933933	-3.868467	-1.249954
259	5/1/99 8:38	1	0.892311	-13.332664	-10.006578
260	5/1/99 10:50	0	0.020062	-3.987897	-1.248529

```
After shifting
```

	time	y	x1	x2	x3
255	5/1/99 8:30	0.0	0.997107	-3.865720	-1.133779
256	5/1/99 8:32	0.0	1.016235	-4.058394	-1.097158
257	5/1/99 8:34	1.0	1.005602	-3.876199	-1.074373
258	5/1/99 8:36	1.0	0.933933	-3.868467	-1.249954
260	5/1/99 10:50	0.0	0.020062	-3.987897	-1.248529

Se notiamo qui, abbiamo spostato l'etichetta positiva al 5/1/99 8:38 su n -1 e n -2 timestamp e abbiamo lasciato cadere la riga n . Inoltre, esiste una differenza di tempo di oltre 2 minuti tra una riga di interruzione e la riga successiva. Questo perché, quando si verifica un'interruzione, la macchina rimane nello stato di interruzione per un po '. Durante questo periodo, abbiamo y = 1 per le righe consecutive. Nei dati forniti, queste righe di interruzione consecutive vengono eliminate per impedire al classificatore di imparare a prevedere un'interruzione **dopo** che è già avvenuta. Fare riferimento a [2] per i dettagli.

Prima di andare avanti, puliamo i dati facendo cadere il tempo e altre due colonne categoriali.

```
# Rimuovi la colonna del tempo e le colonne categoriali
 df = df.drop (['time', 'x28', 'x61'], axis = 1)
```

Preparare i dati di input per LSTM

LSTM è un po 'più impegnativo rispetto ad altri modelli. Una notevole quantità di tempo e attenzione può essere dedicata alla preparazione dei dati adatti a un LSTM. Tuttavia, in genere vale la pena.

I dati di input per un modello LSTM sono un array tridimensionale. La forma della matrice è *campioni* x *lookback* x *caratteristiche* . Comprendiamoli,

- *campioni* : questo è semplicemente il numero di osservazioni o, in altre parole, il numero di punti dati.
- *lookback* : i modelli LSTM sono pensati per guardare al passato. Significato, al momento *t* l'LSTM elaborerà i dati fino a (*t* - *lookback*) per fare una previsione.
- *caratteristiche* : è il numero di funzioni presenti nei dati di input.

Innanzitutto, estrarremo le funzionalità e la risposta.

```
input_X = df.loc [:, df.columns! = 'y']. valori   # converte il df
in un array
 numpy input_y = df ['y']. valori

n_features = input_X.shape [1]   # numero di funzioni
```

Il `input_X` qui è una matrice a 2 dimensioni del formato *campioni* x *caratteristiche* . Vogliamo essere in grado di trasformare un tale array 2D in un array 3D di dimensioni: *campioni* x *lookback* x *caratteristiche* . Fare riferimento alla Figura 1 sopra per una comprensione visiva.

Per questo, sviluppiamo una funzione `temporalize` .

```
def temporalizzare (X, Y, analisi):
    X = []
    y = []
    per i in range (len (input_X) -lookback-1):
```

```
        t = []
        per j in serie (1, lookback + 1):
            # Raccogli i record passati fino al periodo di ricerca
             t.append (input_X [[(i + j + 1)]
        ,: ]) X.append (t)
        y.append (input_y [i + lookback + 1])
    return X, y
```

Per testare e dimostrare questa funzione, vedremo un esempio di seguito con `lookback = 5`.

```
print ('Prima istanza di y = 1 nei dati originali')
display (df.iloc [(np.where (np.array (input_y) == 1) [0] [0] -5) :(
np.where (np.array (input_y) == 1) [0] [0] +1),])

lookback = 5 # Equivalente a 10 min di dati passati.
# Temporalizza i dati
X, y = temporalizza (X = input_X, y = input_y, lookback = lookback)

print ('Per la stessa istanza di y = 1, stiamo conservando oltre 5
campioni nell'array di predittori 3D, X.')
display (pd.DataFrame (np.concatenate (X [np.where (np.array (y) ==
1) [0] [0]], axis = 0)))
```

First instance of y = 1 in the original data

	y	x1	x2	x3	x4	x5	x6	x7	x8	x9	...	x51	x52	x53	x54	
252	0.0	0.987078	-4.025989	-1.210205	0.899603	0.450338	14.098854	0.000732	-0.051043	-0.059966	...	29.984624	11.248703	-0.752385	-5.014893	-67.454
253	0.0	0.921726	-3.728572	-1.230373	-1.598718	0.227178	14.594612	0.000061	-0.051043	-0.040129	...	29.984624	11.253342	-0.752385	-5.014987	-58.029
254	0.0	0.975947	-3.913736	-1.304682	0.561987	0.004034	14.630532	0.000732	-0.051043	-0.040129	...	29.984624	11.257736	-0.752385	-5.015081	-61.783
255	0.0	0.997107	-3.865720	-1.133779	0.377295	-0.219126	14.666420	0.000732	-0.061114	-0.040129	...	29.984624	11.262375	-0.752385	-5.015176	-70.151
256	0.0	1.016235	-4.058394	-1.097158	2.327307	-0.442286	14.702309	0.000732	-0.061114	-0.040129	...	29.984624	11.267013	-0.752385	-5.015270	-60.884
257	1.0	1.005602	-3.876199	-1.074373	0.844397	-0.553050	14.738228	0.000732	-0.061114	-0.030057	...	29.984624	11.271652	-0.752385	-5.015364	-69.553

6 rows × 60 columns

For the same instance of y = 1, we are keeping past 3 samples in the 3D predictor array, X.

	0	1	2	3	4	5	6	7	8	9	...	49	50	51	52	
0	0.921726	-3.728572	-1.230373	-1.598718	0.227178	14.594612	0.000061	-0.051043	-0.040129	0.001791	...	29.984624	11.253342	-0.752385	-5.014987	-58
1	0.975947	-3.913736	-1.304682	0.561987	0.004034	14.630532	0.000732	-0.051043	-0.040129	0.001791	...	29.984624	11.257736	-0.752385	-5.015081	-61
2	0.997107	-3.865720	-1.133779	0.377295	-0.219126	14.666420	0.000732	-0.061114	-0.040129	0.001791	...	29.984624	11.262375	-0.752385	-5.015176	-70
3	1.016235	-4.058394	-1.097158	2.327307	-0.442286	14.702309	0.000732	-0.061114	-0.040129	0.001791	...	29.984624	11.267013	-0.752385	-5.015270	-60
4	1.005602	-3.876199	-1.074373	0.844397	-0.553050	14.738228	0.000732	-0.061114	-0.030057	0.001791	...	29.984624	11.271652	-0.752385	-5.015364	-69

Quello che stiamo cercando qui è,

- Nei dati originali, y = 1 alla riga 257.
- Con `lookback = 5` Vogliamo che LSTM guardi le 5 righe prima della riga 257 (incluso se stesso).

(incluso se stesso).

- Nell'array 3D `X`, ciascun blocco 2D `X[i,:,:]` indica i dati di previsione corrispondenti `y[i]`. Per tracciare un'analogia, nella regressione `y[i]` corrisponde un vettore 1D `X[i,:]`; in LSTM `y[i]` corrisponde a un array 2D `X[i,:,:]`.
- Questo blocco 2D `X[i,:,:]` dovrebbe avere i predittori su `input_X[i,:]` e le righe precedenti fino al dato `lookback`.
- Come possiamo vedere nell'output sopra, il `X[i,:,:]` blocco in fondo è uguale alle cinque righe precedenti di $y = 1$ mostrate in alto.
- Allo stesso modo, questo viene applicato per tutti i dati, per tutti gli y. L'esempio qui è mostrato per un'istanza di $y = 1$ per una visualizzazione più semplice.

Dividi in treno, valido e prova

Questo è semplice con la `sklearn` funzione.

```
X_train, X_test, y_train, y_test = train_test_split (np.array (X),
np.array (y), test_size = DATA_SPLIT_PCT, random_state = SEED)

X_train, X_valid, y_train, y_valid = train_tin_t__pl_s_t
random_state = SEED)
```

Per addestrare il codificatore automatico, useremo la X proveniente solo dai dati con etichetta negativa. Pertanto, separiamo la X corrispondente a $y = 0$.

```
X_train_y0 = X_train [y_train == 0]
X_train_y1 = X_train [y_train == 1]

X_valid_y0 = X_valid [y_valid == 0]
X_valid_y1 = X_valid [y_valid == 1]
```

Noi rimodellare le X nella dimensione 3D richiesti: *campione* x *lookback* x *caratteristiche* .

```
X_train = X_train.reshape (X_train.shape [0], lookback, n_features)
X_train_y0 = X_train_y0.reshape (X_train_y0.shape [0], lookback,
n_features)
X_train_y1 = X_train_y1.res )
```

```
X_valid = X_valid.reshape (X_valid.shape [0], lookback, n_features)
X_valid_y0 = X_valid_y0.reshape (X_valid_y0.shape [0], lookback,
n_features)
X_valid_y1 = X_valid_y1 = X_valid_y1 = X n_features)

X_test = X_test.reshape (X_test.shape [0], lookback, n_features)
```

Standardizzare i dati

Di solito è meglio usare un dato standardizzato (trasformato in gaussiano con media 0 e deviazione standard 1) per gli autocodificatori.

> *Un errore di standardizzazione comune è: normalizziamo tutti i dati e li dividiamo in test di treno. Questo non è corretto I dati dei test dovrebbero essere completamente invisibili a qualsiasi cosa durante la modellazione. Pertanto, dovremmo normalizzare i dati di addestramento e utilizzare le sue statistiche riassuntive per normalizzare i dati del test (per la normalizzazione, queste statistiche sono la media e le varianze di ciascuna caratteristica).*

La standardizzazione di questi dati è un po 'complicata. Questo perché le matrici X sono 3D e vogliamo che la standardizzazione avvenga rispetto ai dati 2D originali.

Per fare questo, avremo bisogno di due UDF.

- `flatten` : Questa funzione consente di ricreare l'array 2D originale da cui sono stati creati gli array 3D. Questa funzione è l'inverso del `temporalize` significato `X = flatten(temporalize(X))` .
- `scale` : Questa funzione ridimensionerà un array 3D che abbiamo creato come input per LSTM.

```
def flatten (X):
    ''' '
    Appiattisci un array 3D.     Input     XA array 3D per lstm,
dove l'array è campione x timesteps x funzionalità.     Output
flattened_X Un array 2D, caratteristiche di esempio x.     ''' '
flattened_X = np.empty ((X.shape [0], X.shape [2]))   # sample x
features array. per i in range (X.shape [0]):         flattened_X
[i] = X [i, (X.shape [1] -1),:] ritorno (flattened_X) def scala (X,
ablatore): '''     Scala 3D array.     Input     XA array 3D per
lstm, in cui l'array è campione x timesteps x caratteristiche.
```

```
    scaler Un oggetto scaler, ad esempio
sklearn.preprocessing.StandardScaler,
sklearn.preprocessing.normalize     Output     3D Scaled array 3D.
''' Per i in range (X.shape [0]):         X [i,:,:] =
scaler.transform (X [i,:,:]) ritorno X
```

> *Perché non abbiamo prima normalizzato i dati 2D originali e quindi creato le matrici 3D? Perché, per fare ciò, dovremo: suddividere i dati in treno e test, seguiti dalla loro normalizzazione. Tuttavia, quando creiamo le matrici 3D sui dati di test, perdiamo le righe iniziali dei campioni fino alla* ricerca. *La suddivisione in test del treno valido provocherà questo sia per la validazione che per i set di test.*

Adatteremo un oggetto di standardizzazione da `sklearn` . Questa funzione standardizza i dati su Normale (0, 1). Si noti che è necessario appiattire l' `X_train_y0` array per passare alla `fit` funzione.

```
# Inizializza uno scaler utilizzando i dati di allenamento.
scaler = StandardScaler (). fit (appiattire (X_train_y0))
```

Useremo il nostro UDF `scale` per standardizzare `X_train_y0` l'oggetto trasformato montato `scaler` .

```
X_train_y0_scaled = scale (X_train_y0, scaler)
```

Assicurarsi che abbia `scale` funzionato correttamente?

Una corretta trasformazione di `X_train` assicurerà che i mezzi e le varianze di ciascuna colonna dell'appiattito `X_train` siano rispettivamente 0 e 1. Lo testiamo.

```
a = appiattisci (X_train_y0_scaled)
print ('colwise mean', np.mean (a, axis = 0) .round (6))
print ('colwise variance', np.var (a, axis = 0))
```

```
colwise mean [ 0. -0.  0.  0.  0. -0.  0.  0.  0.  0.  0. -0. -0. -0. -0.  0. -0. -0.
 -0. -0.  0.  0. -0. -0. -0.  0.  0.  0. -0.  0.  0.  0. -0. -0. -0.  0.
  0.  0. -0. -0.  0.  0. -0. -0.  0.  0.  0.  0.  0.  0.  0. -0. -0.  0.
  0. -0. -0.  0.  0.]
colwise variance [1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.
 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.
 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]
```

Tutti i mezzi e le varianze emessi sopra sono rispettivamente 0 e 1. Pertanto, il ridimensionamento è corretto. Ora scaleremo i set di validazione e test. Useremo nuovamente l' `scaler` oggetto su questi set.

```
X_valid_scaled = scale (X_valid, scaler)
X_valid_y0_scaled = scale (X_valid_y0, scaler)

X_test_scaled = scale (X_test, scaler)
```

Addestramento al codificatore automatico LSTM

Inizializzeremo innanzitutto alcune variabili.

```
timesteps = X_train_y0_scaled.shape [1] # uguale alla ricerca
 n_features = X_train_y0_scaled.shape [2] # 59

 epoche = 200
batch = 64
lr = 0.0001
```

Ora sviluppiamo un'architettura semplice.

```
lstm_autoencoder = Sequential ()
# Encoder
 lstm_autoencoder.add (LSTM (32, activation = 'relu', input_shape =
(timesteps, n_features), return_sequences = True ))
lstm_autoencoder.add (LSTM (16, activation = 'relu',
return_sequences = Falso ))
lstm_autoencoder.add (RepeatVector (timesteps))
# Decoder
 lstm_autoencoder.add (LSTM (16, activation = 'relu',
return_sequences = True ))
lstm_autoencoder.add (LSTM (32, activation = 'relu',
return_sequences = True ))
lstm_autoencoder.add (TimeDistributed (Dense (n_features)))

lstm_autoencoder.summary ()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
lstm_23 (LSTM)               (None, 5, 32)             11776
_________________________________________________________________
lstm_24 (LSTM)               (None, 16)                3136
_________________________________________________________________
repeat_vector_7 (RepeatVecto (None, 5, 16)             0
_________________________________________________________________
lstm_25 (LSTM)               (None, 5, 16)             2112
_________________________________________________________________
lstm_26 (LSTM)               (None, 5, 32)             6272
_________________________________________________________________
time_distributed_6 (TimeDist (None, 5, 59)             1947
=================================================================
Total params: 25,243
Trainable params: 25,243
Non-trainable params: 0
_________________________________________________________________
```

Dal riepilogo (), il numero totale di parametri è 5.331. Questa è circa la metà della dimensione dell'allenamento. Quindi, questo è un modello appropriato per adattarsi. Per avere un'architettura più grande, avremo bisogno di aggiungere regolarizzazione, ad esempio Dropout, che sarà trattato nel prossimo post.

Ora addestreremo il codificatore automatico.

```
adam = optimizers.Adam (lr)
lstm_autoencoder.compile (loss = 'mse', optimizer = adam)

cp = ModelCheckpoint (filepath = "lstm_autoencoder_classifier.h5",
                                 save_best_only = True ,
                                 verbose = 0)

tb = TensorBoard (log_dir = '. / logs ',
                  histogram_freq = 0,
```

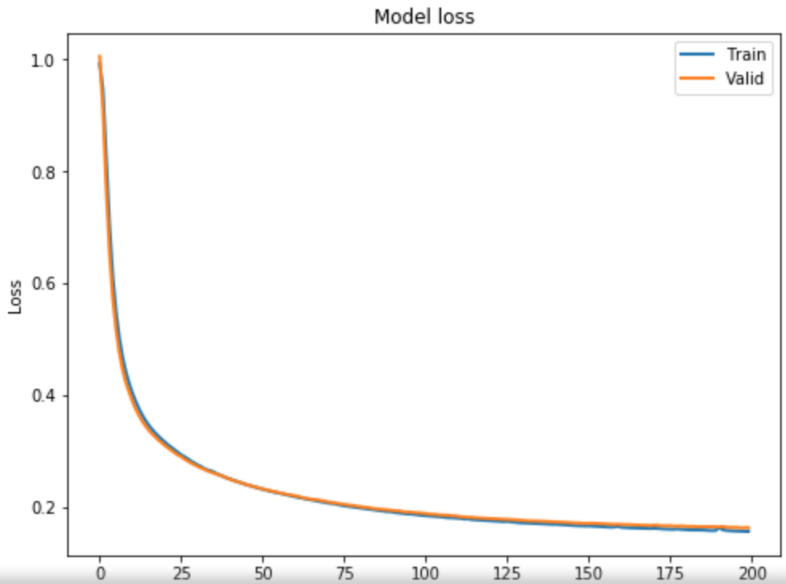
```
                    write_graph = True ,
                    write_images = True )
```

```
lstm_autoencoder_history = lstm_autoencoder.fit (X_train_y0_scaled,
X_train_y0_scaled
                                                 ,
                                                 epochs = epochs ,
batch_size = batch,
                                                 validation_data =
(X_valid_y0_scaled, X_valid_y0_scaled),
                                                 verbose = 2)
.history
```

```
Train on 11691 samples, validate on 2923 samples
Epoch 1/500
 - 8s - loss: 0.9955 - val_loss: 1.0305
Epoch 2/500
 - 2s - loss: 0.9949 - val_loss: 1.0282
Epoch 3/500
 - 2s - loss: 0.9709 - val_loss: 0.9869
Epoch 4/500
 - 2s - loss: 0.9382 - val_loss: 0.9601
Epoch 5/500
 - 2s - loss: 0.9099 - val_loss: 0.9326
```

Tracciare il cambiamento nella perdita nel corso delle epoche.

```
plt.plot (lstm_autoencoder_history ['loss'], linewidth = 2, label =
'Train')
plt.plot (lstm_autoencoder_history ['val_loss'], linewidth = 2,
label = 'Valid')
plt.legend (loc = 'upper right ')
plt.title (' Perdita modello ')
plt.ylabel (' Perdita ')
plt.xlabel (' Epoch ')
plt.show ()
```



Epoch

Figura 2. Funzione di perdita sulle epoche.

Classificazione

Simile al precedente post [1], qui mostriamo come possiamo usare un errore di ricostruzione di Autoencoder per la classificazione degli eventi rari. Seguiamo questo concetto: ci si aspetta che il codificatore automatico ricostruisca un rumore se l'errore di ricostruzione è elevato, lo classificheremo come una rottura del foglio.

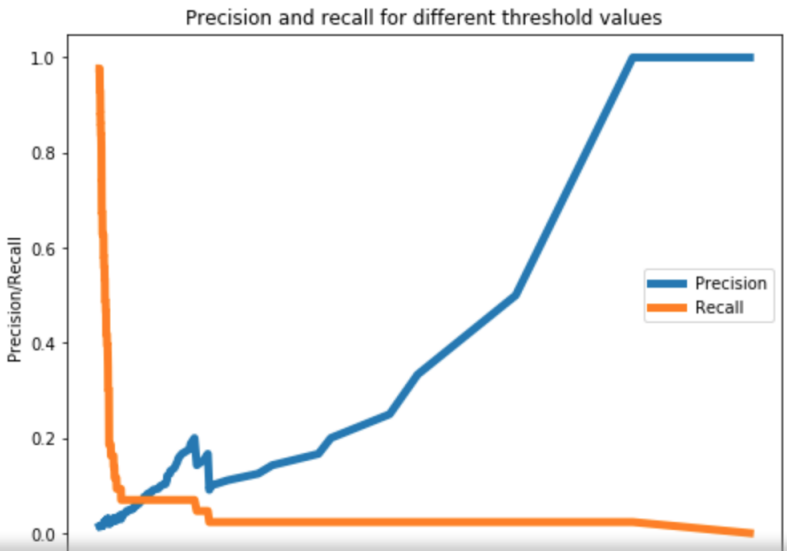
Dovremo determinare la soglia per questo. Inoltre, tieni presente che qui utilizzeremo l'intero set di convalida contenente sia y = 0 o 1.

```
valid_x_predictions = lstm_autoencoder.predict (X_valid_scaled)
MSE = np.mean (np.power (appiattire (X_valid_scaled) - appiattire
(valid_x_predictions), 2), asse = 1)

error_df = pd.DataFrame ({ 'Reconstruction_error': MSE,
                           'True_class ': y_valid.tolist ()})

precision_rt, remind_rt ,reshold_rt = precision_recall_curve
(error_df.True_class, error_df.Reconstruction_error)
plt.plot (reshold_rt, precision_rt [1:], label = "Precision",
linewidth = 5)
plt.pl (reshold_rt, remind_rt [1:], label = "Recall", linewidth = 5)
plt.title ('Precisione e richiamo per valori soglia diversi')
plt.xlabel ('Soglia')
plt.ylabel ('Precisione / Richiamo' )
plt.legend ()
plt.show ()
```

Nota che dobbiamo `flatten` calcolare le matrici `mse` .



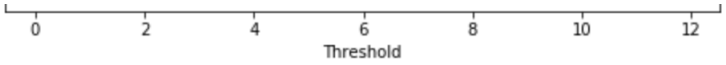


Figura 3. Una soglia di 0,3 dovrebbe fornire un ragionevole compromesso tra precisione e richiamo, poiché vogliamo un richiamo più elevato.

Ora eseguiremo la classificazione sui dati del test.

> ***Non dovremmo stimare la soglia di classificazione dai dati del test. Ciò comporterà un eccesso di adattamento.***

```
test_x_predictions = lstm_autoencoder.predict (X_test_scaled)
MSE = np.mean (np.power (appiattire (X_test_scaled) - appiattire
(test_x_predictions), 2), asse = 1)

error_df = pd.DataFrame ({ 'Reconstruction_error': MSE,
                           'True_class ': y_test.tolist ()}
)reshold_fixed

= 0.3
groups = error_df.groupby (' True_class ')
fig, ax = plt.subplots ()

per nome, gruppo in gruppi:
    ax.plot (group.index, group.Reconstruction_error, marker = 'o',
ms = 3.5, linestyle = '',
            label = "Break" se name == 1 else "Normal")
ax.hlines (soglia_fissa, ax.get_xlim () [0], ax.get_xlim () [1],
colors = "r",zorder = 100, label = 'Soglia')
ax.legend ()
plt.title ("Errore di ricostruzione per classi diverse")
plt.ylabel ("Errore di ricostruzione")
plt.xlabel ("Indice punti dati")
plt.show ();
```

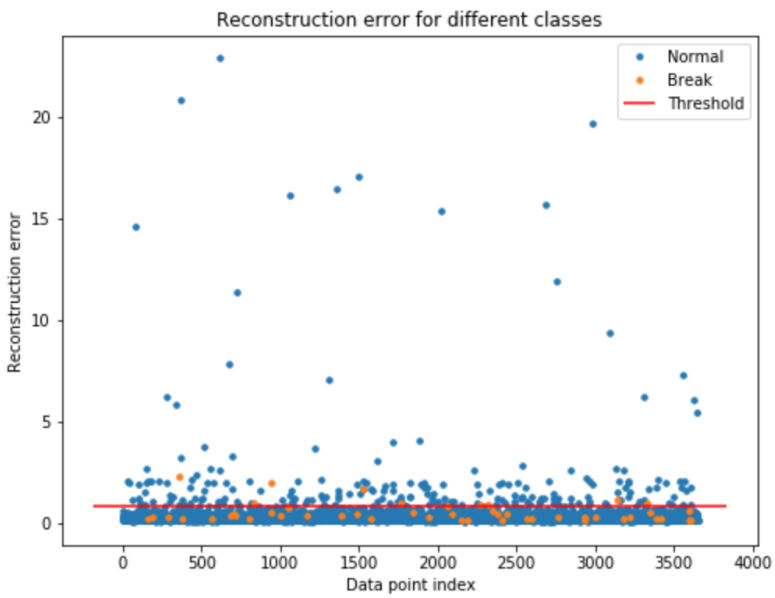


Figura 4. Utilizzo della soglia = 0,8 per la classificazione. I punti arancione e blu sopra la linea di soglia rappresentano rispettivamente il vero positivo e il falso positivo.

Nella Figura 4, il punto arancione e blu sopra la linea di soglia rappresenta rispettivamente il vero positivo e il falso positivo. Come possiamo vedere, abbiamo un buon numero di falsi positivi.

Vediamo i risultati della precisione.

Precisione del test

Matrice di confusione

```
pred_y = [1 if e> soglia_fissata altro 0 per e in
error_df.Reconstruction_error.values]

conf_matrix = confusion_matrix (error_df.True_class, pred_y)

plt.figure (figsize = (6, 6))
sns.heatmap (conf_matrix, xticklabels = LAB , yticklabels = LABELS,
annot = True , fmt = "d");
plt.title ("Matrice di confusione")
plt.ylabel ('True class')
plt.xlabel ('Classe prevista')
plt.show ()
```

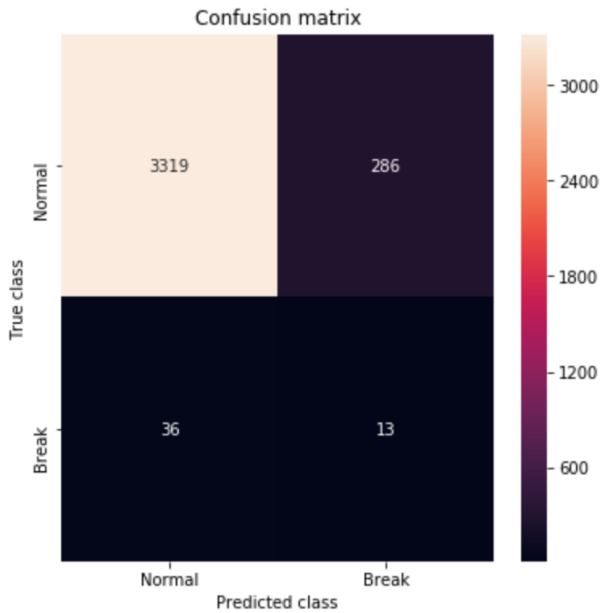


Figura 5. Matrice di confusione che mostra i veri positivi e i falsi positivi.

ROC Curve e AUC

```
false_pos_rate, true_pos_rate, soglie = roc_curve
(error_df.True_class, error_df.Reconstruction_error)
roc_auc = AUC (false_pos_rate, true_pos_rate,)

plt.plot (false_pos_rate, true_pos_rate, linewidth = 5, label = 'AUC
= % 0.3f ' % roc_auc)
plt .plot ([0,1], [0,1], linewidth = 5)

plt.xlim ([- 0.01, 1])
plt.ylim ([0, 1.01])
plt.legend (loc = 'in basso a destra' )
plt.title ("Curva caratteristica operativa del ricevitore (ROC)")
plt.ylabel ("Tasso positivo reale")
plt.xlabel ("Tasso positivo falso")
plt.show ()
```

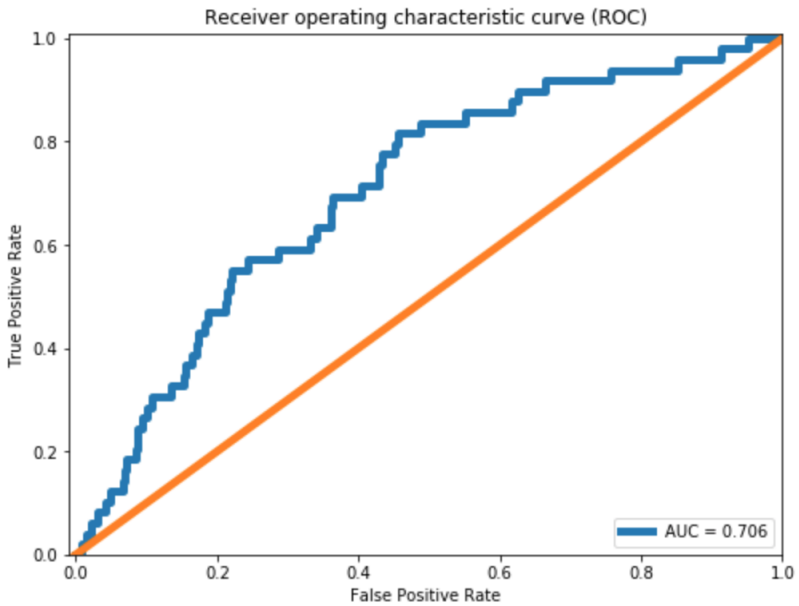


Figura 6. La curva ROC.

Vediamo un miglioramento di circa il 10% nell'AUC rispetto al denso strato Autoencoder in [1]. Dalla matrice di confusione nella Figura 5, abbiamo potuto prevedere 10 delle 39 istanze di interruzione. Come discusso anche in [1], questo è significativo per una cartiera. Tuttavia, il miglioramento che abbiamo ottenuto rispetto al denso strato Autoencoder è minore.

Il motivo principale è che il modello LSTM ha più parametri da stimare. Diventa importante utilizzare la regolarizzazione con gli LSTM. La regolarizzazione e altri miglioramenti del modello saranno discussi nel prossimo post.

Repository Github

cran2367 / lstm_autoencoder_classifier
Un codificatore automatico LSTM per la classificazione di eventi rari. Contribuisci allo...

github.com

Cosa si può fare di meglio?

Nel prossimo articolo, impareremo a sintonizzare un Autoencoder. Andremo oltre

- CNN LSTM Autoencoder,
- Livello di abbandono,
- Dropout LSTM (Dropout_U e Dropout_W)
- Livello di abbandono gaussiano
- Attivazione SELU e
- alpha-dropout con attivazione SELU.

Conclusione

Questo post ha continuato il lavoro sui dati con etichetta binaria di eventi rari estremi in [1]. Per utilizzare i modelli temporali, i codificatori automatici LSTM vengono utilizzati per creare un classificatore di eventi rari per un processo multivariato di serie temporali. Vengono discussi i dettagli sulle fasi di preelaborazione dei dati per il modello LSTM. Un semplice modello di codificatore automatico LSTM viene addestrato e utilizzato per la classificazione. È stato riscontrato un certo miglioramento della precisione rispetto a un denso autoencoder. Per un ulteriore miglioramento, esamineremo i modi per migliorare un Autoencoder con Dropout e altre tecniche nel prossimo post.

. . .

Follow-up Leggi

Si consiglia di leggere la comprensione passo-passo dei livelli di LSTM Autoencoder per cancellare i concetti di rete LSTM.

Riferimenti

1. Classificazione di eventi rari estremi utilizzando Autoencoder in Keras
2. Ranjan, C., Mustonen, M., Paynabar, K., & Pourak, K. (2018). Set di dati: classificazione degli eventi rari nelle serie temporali multivariate. *arXiv prestampa arXiv: 1809.10717*
3. Previsioni di serie temporali con deep learning e autoencoder LSTM
4. Codice completo: LSTM Autoencoder

Dichiarazione di non responsabilità: lo scopo di questo post è limitato a un tutorial per la creazione di un codificatore automatico LSTM e il suo utilizzo come classificatore di eventi rari. Ci si aspetta che un professionista ottenga risultati migliori per questi dati mediante la messa a punto della rete. Lo scopo di questo articolo è aiutare i data scientist a implementare un codificatore automatico LSTM.