TIME-SERIES NEURAL NETWORK FOR FORECASTING BUILDING ENERGY CONSUMPTION

Laura Hwa

TABLE OF CONTENTS

INSPIRATION

PSet 6 & 7 - From UCI ML Repo

Energy efficiency Data Set

Download: Data Folder, Data Set Description

Abstract: This study looked into assessing the heating load and cooling load requirements of buildings (that is, energy efficiency) as a function of building parameters.

Data Set Characteristics:	Multivariate	**Number of Instances:**	768	**Area:**	Computer
Attribute Characteristics:	Integer, Real	**Number of Attributes:**	8	**Date Donated**	2012-11-30
Associated Tasks:	Classification, Regression	**Missing Values?**	N/A	**Number of Web Hits:**	410571

Source:

The dataset was created by Angeliki Xifara (angxifara '@' gmail.com, Civil/Structural Engineer) and was processed by Athanasios Tsanas (tsanasthanasis '@' gmail.com, Oxford Centre for Industrial and Applied Mathematics, University of Oxford, UK).

Data Set Information:

We perform energy analysis using 12 different building shapes simulated in Ecotect. The buildings differ with respect to the glazing area, the glazing area distribution, and the orientation, amongst other parameters. We simulate various settings as functions of the afore-mentioned characteristics to obtain 768 building shapes. The dataset comprises 768 samples and 8 features, aiming to predict two real valued responses. It can also be used as a multi-class classification problem if the response is rounded to the nearest integer.

DATA

In this competition, you'll develop accurate models of metered building energy usage in the following areas: chilled water, electric, hot water, and steam meters. The data comes from over 1,000 buildings over a three-year timeframe. With better estimates of these energy-saving investments, large scale investors and financial institutions will be more inclined to invest in this area to enable progress in building efficiencies.

DATA

Evaluation Metric

The evaluation metric for this competition is Root Mean Squared Logarithmic Error.

The RMSLE is calculated as

$$\epsilon = \sqrt{\frac{1}{n} \sum_{i=1}^{n} (\log(p_i + 1) - \log(a_i + 1))^2}$$

DATA

This competition challenges you to build predictive models for electricity consumption based on building metadata, historic usage, and weather data. The dataset includes hourly meter readings from 100 buildings at several different sites around the world.

train.csv / test.csv

- id - a combination of building_id and time_stamp (only present in test.csv)
- building_id - ID of the building.
- timestamp - When the measurement was taken.
- primary_use - Indicator of the primary category of activities for the building based on EnergyStar property type definitions.
- square_feet - Gross floor area of the building.
- year_built - Year building was opened.
- floor_count - Number of floors of the building.
- air_temperature - Degrees Celsius.
- cloud_coverage - Portion of the sky covered in clouds, in oktas.
- dew_temperature - Degrees Celsius.
- precip_depth_1_hr - Millimeters.
- sea_level_pressure - Millibar/hectopascals.
- wind_direction - Compass direction (0-360).
- wind_speed - Meters per second.
- meter_reading - The target variable. Energy consumption in kWh.

IMPLEMENTATION

DATA PREPARATION

```
#import libraries
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from numpy import array
from keras.models import Seqeuntial
from keras.layers import LSTM
from keras.layers import Dense

#import data file
data = pd.read_csv(r'/Users/laurah/Desktop/code/neural net proj/train.csv')

def parser(x):
    return pd.datetime.strptime('190'+x, '%Y-%m')

#delete meter readings of 0
condition = data["meter_reading"]>1e-10
data_clean = data[condition]
column_names = ["timestamp","meter_reading"]

#examine time and meter reading
data_clean = data_clean[["timestamp", "meter_reading"]]
data_clean = data_clean.astype({"timestamp": str})
data_clean = data_clean.astype({"meter_reading": float})
data_clean.reset_index(drop = True, inplace = True)
```

```
#create new pandas array to populate
newdata = pd.DataFrame(index = range(len(data_clean)), columns = column_names)

for i in range(len(data_clean)):
    new_timestamp = data_clean["timestamp"][i][0:10]
    new_meter = data_clean["meter_reading"][i]
    newdata.loc[i] = [new_timestamp, new_meter]

#print(newdata)

#find data from same day and average
avgdata = pd.DataFrame(index = range(366), columns = column_names)
avgdata = newdata.groupby("timestamp")["meter_reading"].mean()
print(avgdata)

avgdata = avgdata.to_frame()
ts = newdata["timestamp"].unique()
avgdata.insert(0, "timestamp", ts, True)
print(avgdata)
```

THEORY

Long Short-Term Memory networks (LSTMs)

- ➢ Time series forecasting
- ➢ Univariate LSTM Model
  - Model learns from a series of past observations to predict the next value in the sequence
  - Single-step
- ➢ Type of RNN

Arguments

- **units**: Positive integer, dimensionality of the output space.
- **activation**: Activation function to use. Default: hyperbolic tangent (`tanh`). If you pass `None`, no activation is applied (ie. "linear" activation: `a(x) = x`).
- **recurrent_activation**: Activation function to use for the recurrent step. Default: sigmoid (`sigmoid`). If you pass `None`, no activation is applied (ie. "linear" activation: `a(x) = x`).
- **use_bias**: Boolean (default `True`), whether the layer uses a bias vector.
- **kernel_initializer**: Initializer for the `kernel` weights matrix, used for the linear transformation of the inputs. Default: `glorot_uniform`.
- **recurrent_initializer**: Initializer for the `recurrent_kernel` weights matrix, used for the linear transformation of the recurrent state. Default: `orthogonal`.
- **bias_initializer**: Initializer for the bias vector. Default: `zeros`.
- **unit_forget_bias**: Boolean (default `True`). If True, add 1 to the bias of the forget gate at initialization. Setting it to true will also force `bias_initializer="zeros"`. This is recommended in Jozefowicz et al..

Args	
`units`	Positive integer, dimensionality of the output space.
`activation`	Activation function to use. If you don't specify anything, no activation is applied (ie. "linear" activation: `a(x) = x`).
`use_bias`	Boolean, whether the layer uses a bias vector.
`kernel_initializer`	Initializer for the `kernel` weights matrix.
`bias_initializer`	Initializer for the bias vector.
`kernel_regularizer`	Regularizer function applied to the `kernel` weights matrix.
`bias_regularizer`	Regularizer function applied to the bias vector.
`activity_regularizer`	Regularizer function applied to the output of the layer (its "activation").
`kernel_constraint`	Constraint function applied to the `kernel` weights matrix.
`bias_constraint`	Constraint function applied to the bias vector.

IMPLEMENTATION

VANILLA LSTM SETUP

```python
#create batches of three for training
def split_sequence(sequence, n_steps):
    X, y = list(),list()
    for i in range(len(sequence)):
        end_idx = i + n_steps
        if end_idx > len(sequence) - 1:
            break
        seq_x, seq_y = sequence[i:end_idx], sequence[end_idx]
        X.append(seq_x)
        y.append(seq_y)
    return array(X), array(y)

#transform df to array/list
L = avgdata["meter_reading"].astype(float).values.tolist()
seq = L
#choosing my "batch size
n_steps = 3
#split into samples
X, y = split_sequence(seq, n_steps)

for i in range(len(X)):
    print(X[i], y[i])
```

```
[369.11430634 372.72968329 400.59049027] 404.4574066350716б
[372.72968329 400.59049027 404.45740664] 403.099465509904
[400.59049027 404.45740664 403.09946551] 403.215835423656
[404.45740664 403.09946551 403.21583542] 394.8792171591497
[403.09946551 403.21583542 394.87921716] 362.8576515104163б
[403.21583542 394.87921716 362.85765151] 365.3079360840597б
[394.87921716 362.85765151 365.30793608] 405.9013353466386З
[362.85765151 365.30793608 405.90133535] 407.1929332991279
[365.30793608 405.90133535 407.1929333 ] 396.50313933996796
[405.90133535 407.1929333  396.50313934] 347.296568707482З
[407.1929333  396.50313934 347.29656871] 333.9284730867998
[396.50313934 347.29656871 333.92847309] 330.1142059004621
[347.29656871 333.92847309 330.1142059 ] 345.1646184873957
[333.92847309 330.1142059  345.16461849] 380.9064657407408
[330.1142059  345.16461849 380.90646574] 399.1863428197067
[345.16461849 380.90646574 399.18634282] 404.1146188197769
[380.90646574 399.18634282 404.11461882] 407.6983155925152
[399.18634282 404.11461882 407.69831559] 387.25370419506925
[404.11461882 407.69831559 387.2537042 ] 367.996196155843б
[407.69831559 387.2537042  367.99619616] 366.42887063409586
[387.2537042  367.99619616 366.42887063] 405.9426812337999
[367.99619616 366.42887063 405.94268123] 406.50011753674596
[366.42887063 405.94268123 406.50011754] 391.9226042631582
[405.94268123 406.50011754 391.92260426] 402.1110292708323
[406.50011754 391.92260426 402.11102927] 389.8627600936529
[391.92260426 402.11102927 389.86276009] 343.2429351774529
[402.11102927 389.86276009 343.24293518] 349.4014188113692
[389.86276009 343.24293518 349.40141881] 381.426356710183З
[343.24293518 349.40141881 381.42635671] 386.1755111932417
```

IMPLEMENTATION

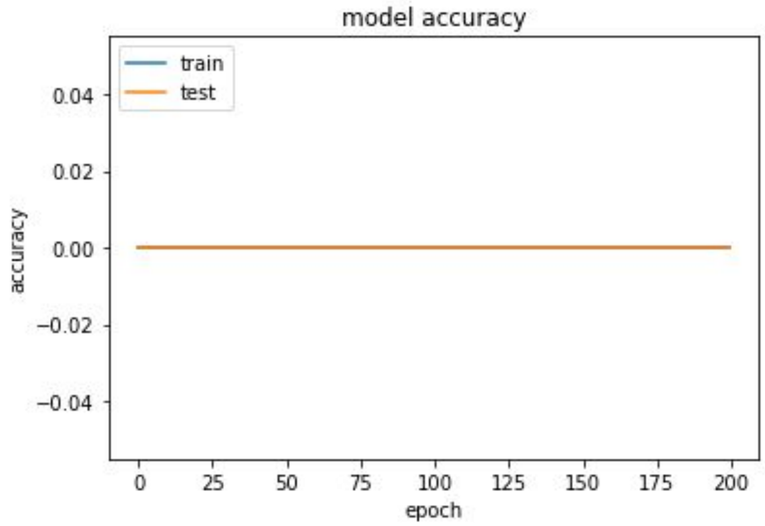
LSTM MODEL

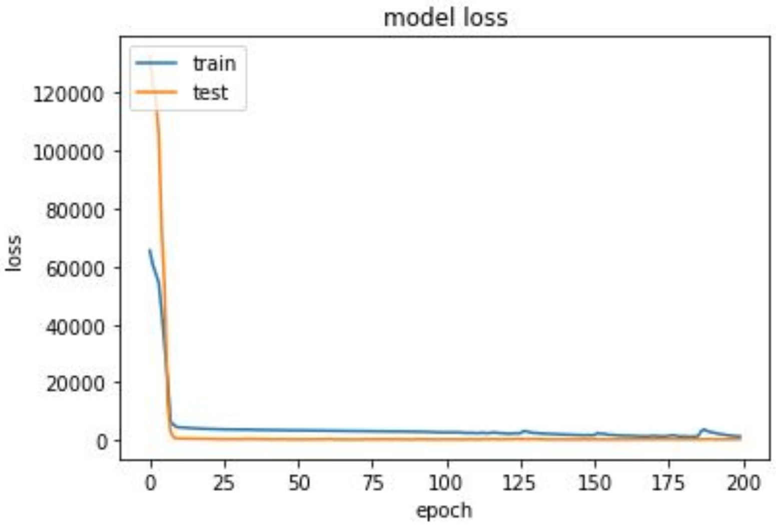
```
# create model
model = keras.models.Sequential()
n_features = 1
model.add(LSTM(50, activation='relu', input_shape=(n_steps, n_features)))
model.add(Dense(1))

#compile model
model.compile(loss = 'mse', optimizer='adam', metrics=['accuracy'])

#reshape from [samples, timestamps] into [samples, timestamps, features]
n_features = 1
X = X.reshape((X.shape[0], X.shape[1], n_features))

#fit model
history = model.fit(X, y, validation_split = 0.33, epochs = 200)

#demonstrate prediction
for i in range(len(X)):
    x_input = X[i]
    x_input = x_input.reshape((1, n_steps, n_features))
    yhat = model.predict(x_input)
    print(yhat)

#list all data in history
print(history.history.keys())
```

```
# summarize history for accuracy
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.show()

# summarize history for loss
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.show()
```

IMPLEMENTATION

LSTM MODEL RESULTS





DATA - PRELIMINARY DIRECTION

Problem

- ➢ No accessible “test” meter_reading values

Solution

- ➢ Self-generate “test” values from taking median instead of average non-zero “train” meter_reading for each “day”

```
#create test df to populate
testdata = pd.DataFrame(index = range(366), columns = column_names)
testdata = newdata.groupby("timestamp")["meter_reading"].median()
print(testdata)

testdata = testdata.to_frame()
ts = testdata["timestamp"].unique()
testdata.insert(0, "timestamp", ts, True)
print("test data")
print(testdata)
```

FUTURE WORK

- Use the self-generated median test data set to validate accuracy
- Tinker with the learning rate, batch size, and other parameters (like the previous MATLAB exercises)

- Explore other LSTMs (ex: stacked)
- Understand more theory behind NNs and ML
- Consider another expansion of the same dataset of using the other building parameters offered to generate a time-series based meter_reading prediction
  - "Feature" focused approach
- Consider another expansion of the UCI dataset
  - Classification focused approach