Data Science Machine Learning and Statistics Solution

March 10, 2019

1 Part 1: Data Analysis

1.1

Import Python libraries and read data.

```
In [40]: from pandas import DataFrame
         from pandas import Series
         from pandas import concat
         from pandas import read_csv
         from pandas import datetime
         from sklearn.metrics import mean_squared_error
         from sklearn.preprocessing import MinMaxScaler
         from keras.models import Sequential
         from keras.layers import Dense, Dropout
         from keras.layers import LSTM
         from math import sqrt
         from matplotlib import pyplot
         from numpy import array
         import json
         import pandas as pd
         import matplotlib.pyplot as plt
         import numpy as np
         import math
         import matplotlib.pyplot as plt
         import  os

         with open('MLandStatsDataSet.json') as json_file:
             data = json.load(json_file)
```

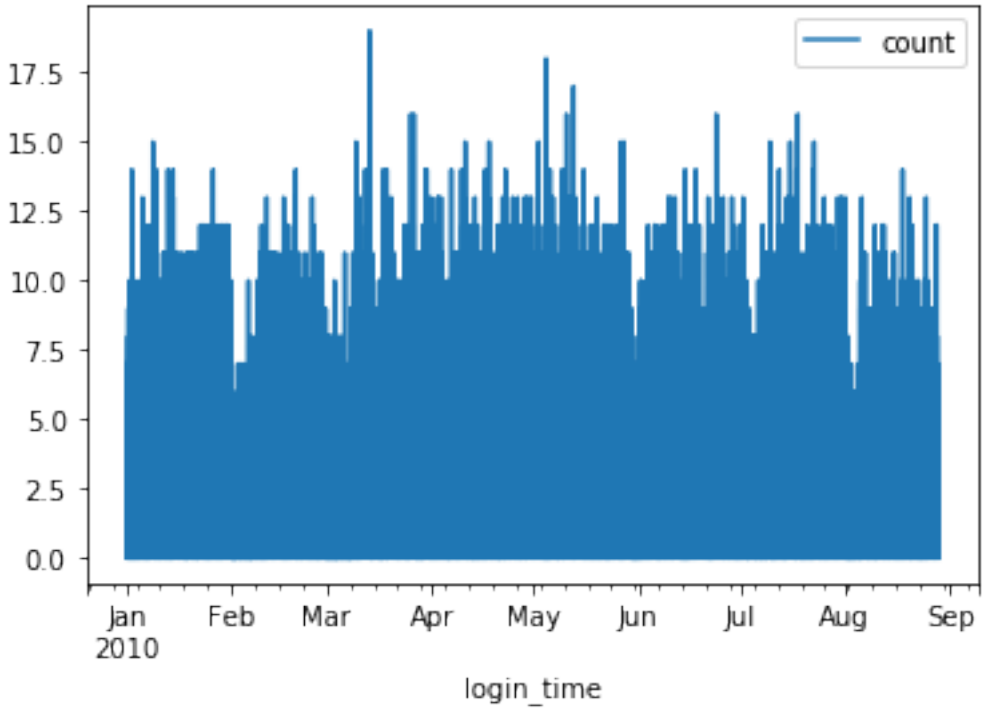
Aggregate the login counts based on 15-minute time intervals and saved as df.

```
In [17]: df = pd.DataFrame(data)
         df['login_time']=pd.to_datetime(df['login_time'])
         gb=df.groupby([pd.Grouper(key='login_time',freq='15min')]).size().to_frame('count')
```

Plot the time series of the login counts:

```
In [18]: gb.plot()
```

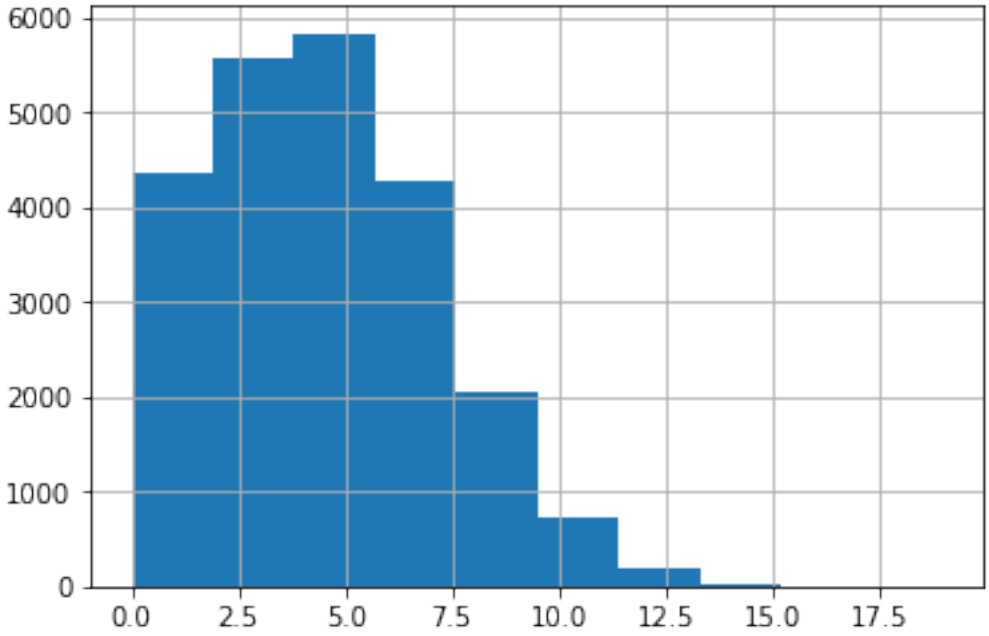
```
Out[18]: <matplotlib.axes._subplots.AxesSubplot at 0x206a65b7d30>
```



Plot the histogram of the login counts:

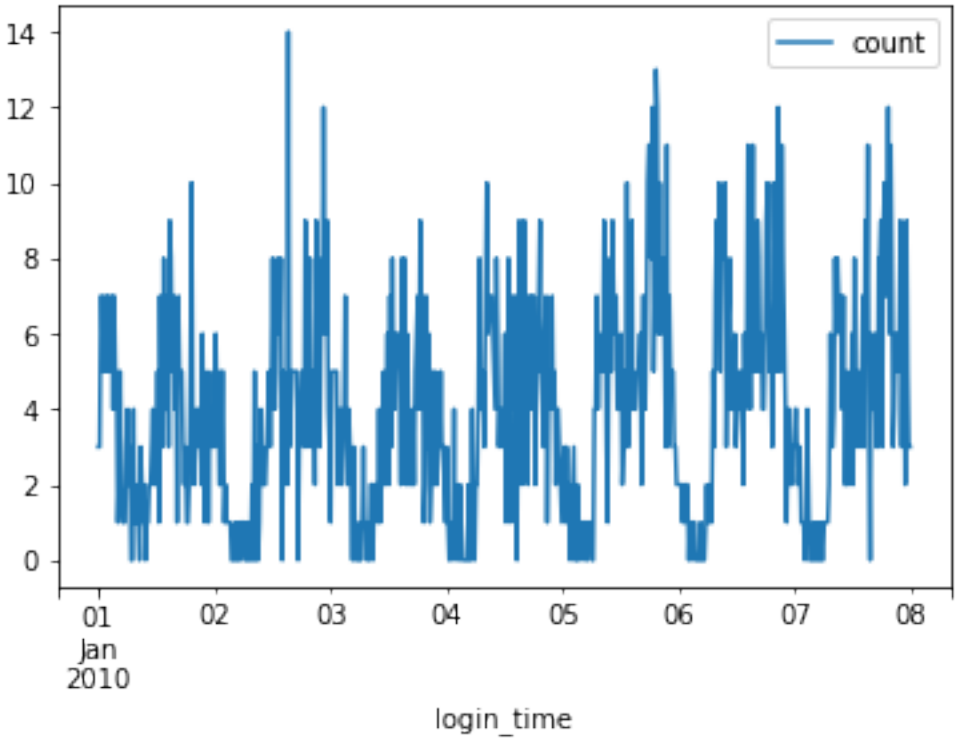
```
In [19]: gb['count'].hist(bins=10)
```

```
Out[19]: <matplotlib.axes._subplots.AxesSubplot at 0x206b15f0b70>
```



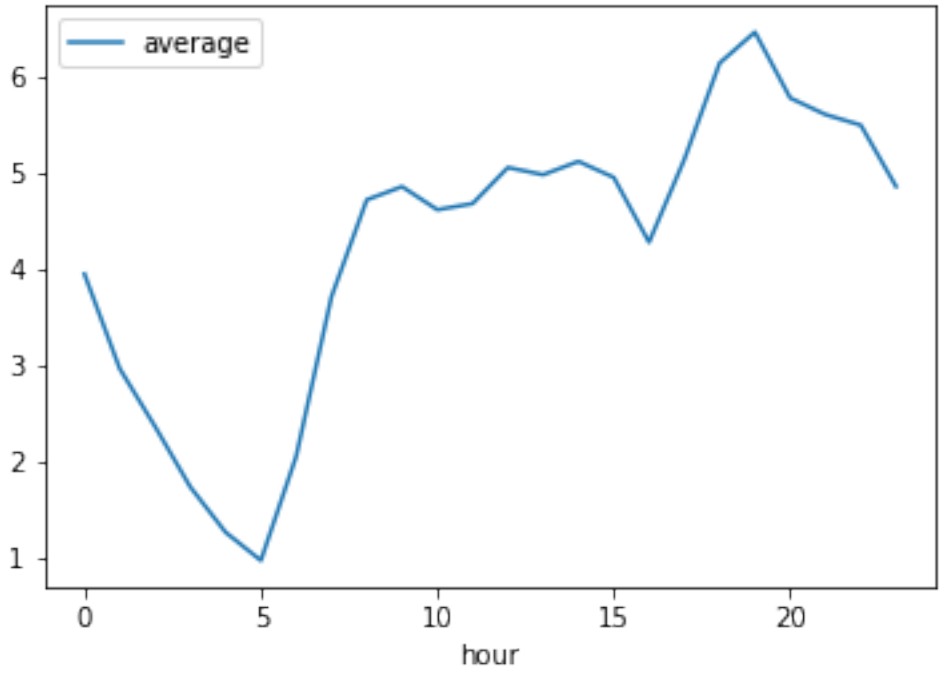
From the time variation plot and the histogram plot of the login counts, we can see that the login count in 15-min time window has values range from 0 to about 18. Most values of login count are less than 8 and no significant outlier exists. Next, we want to check whether there is a daily cycle of the login count. We plot the time series for the first 7 days only:

```
In [20]: gb[0:24*4*7].plot()

Out[20]: <matplotlib.axes._subplots.AxesSubplot at 0x206b1697c50>
```



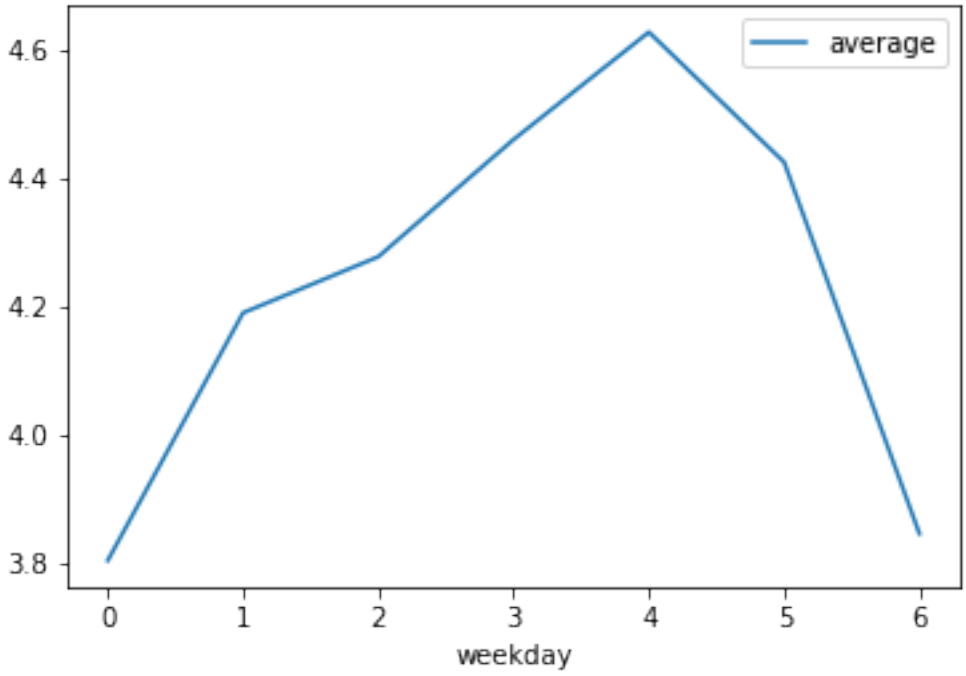
We can clearly see there are daily cycles of the login count. To make sure the daily cycle exists at all the time, we plot the average login count per 15 min for different hours in a day:

```
In [21]: hc = gb.reset_index()
         hc['hour'] = hc['login_time'].dt.hour
         hc=hc.groupby('hour')['count'].agg(['sum','count'])
         hc.reset_index
         hc['average']=hc['sum']/hc['count']
         hc.plot(y='average')

Out[21]: <matplotlib.axes._subplots.AxesSubplot at 0x206b174de10>
```



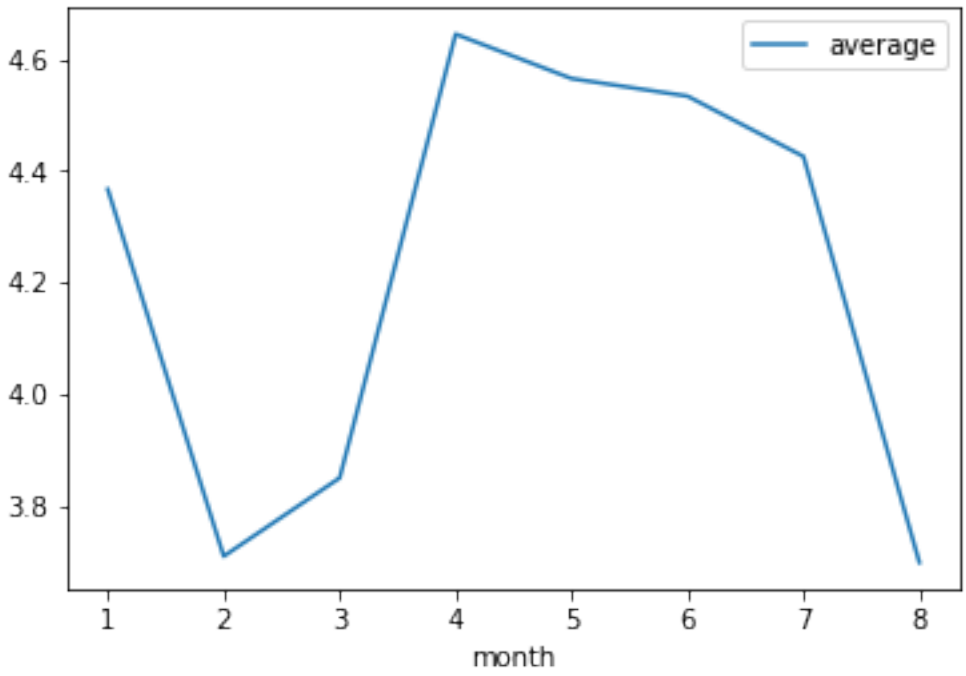
From this plot, we can see that the daily cycle is significant. The login count reaches its minimum value at about 5AM and maximum value at about 6PM. Moreover, we also want to check the dependence of the login count on different days of week. We plot the average login count per 15 min for different days in a week:

```
In [22]: wc = gb.reset_index()
         wc['weekday'] = wc['login_time'].dt.dayofweek
         wc=wc.groupby('weekday')['count'].agg(['sum','count'])
         wc.reset_index
         wc['average']=wc['sum']/wc['count']
         wc.plot(y='average')

Out[22]: <matplotlib.axes._subplots.AxesSubplot at 0x206b17e3f28>
```



From above plot, we can see that the login count is maximum at the day 4 of week (means Friday in Python). Totally, both the daily cycle and weekly cycle match the common sense that the traffic is busiest in the early evening and Friday, which may lead to most guests to use Uber.

```
In [23]: mc = gb.reset_index()
         mc['month'] = mc['login_time'].dt.month
         mc=mc.groupby('month')['count'].agg(['sum','count'])
         mc.reset_index
         mc['average']=mc['sum']/mc['count']
         mc.plot(y='average')

Out[23]: <matplotlib.axes._subplots.AxesSubplot at 0x206b182c668>
```



Finally, we plot the average login count per 15 min for different months in a week as shown above. It seems that the login count is larger at summer. However, only 8 month's data is not enough to analysis the seasonal dependence of the login count.

1.2

To predict time series data, usually we need to consider three contributions: the level, trend and seasonal effect. Plenty of machine learning models can be applied into such kind of prediction. In this report, the model we choose for the prediction is the Long Short-Term Memory (LSTM) model, which is a kind of Recurrent Neural Network model and well-suited to classifying, processing and making predictions based on time series data.

We use the LSTM model in python keras library to process the prediction. First we need transform the origin time series data into data sets for training and predicting. Several parameters need to be set up first:

1.n_lag: the length of time intervals for the input data, considering the significant daily cycle, we set it to be $4 \times 24 = 96$ (one day).

2.n_seq: the length of time intervals for the output prediction, which is set to be 4 according to the requirement of this exercise.

3.n_test: the numbers of samples for the testing, which is set to be $4 \times 24 \times 30 \times 2 = 5760$ (about 2 month).

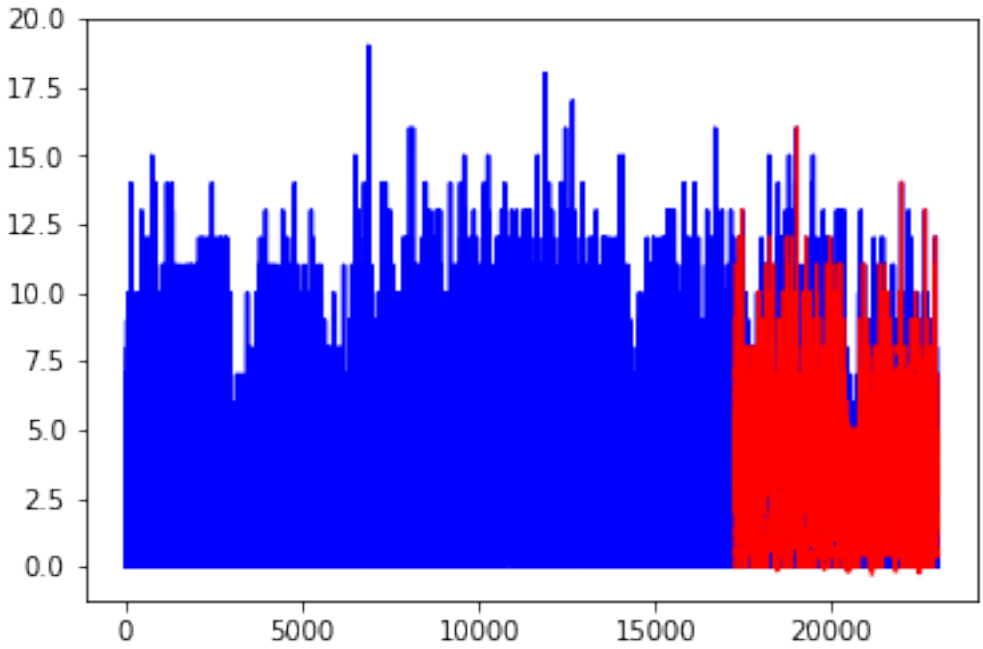
After determining the above parameters, we need to transform the data. First, we perform a normalized process on the data to make the range of data values become $[-1, 1]$ and then obtain the data sets for the model training and testing. The codes of functions used for transforming data as well as training model and making prediction test are list in the Appendix part at the end of this document.

The next step is set up and train the model. We add two hidden layers with 20 neurons in each layer into the model. The batch size is set to be 1 and the number of training epochs is set to be 300. The loss function is chosen to be root mean square error (RMSE) and Adam optimization algorithm is applied.

Finally, we can start to train the model and make predictions through below queries:

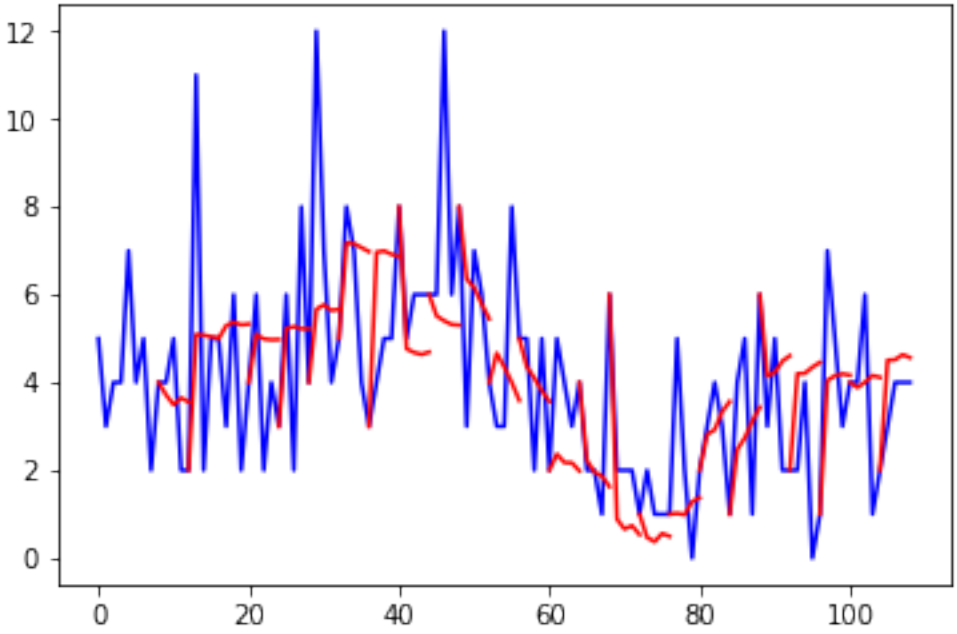
```
In [31]: series  = gb['count']
         series  = series .astype('float32')
         # set up the parameters for the data set and model
         n_lag = 4*24
         n_seq = 4
         n_test = 4*24*30*2
         n_epochs = 300
         n_batch = 1
         n_neurons = 20
         # prepare data
         scaler, train, test = prepare_data(series, n_test, n_lag, n_seq)
         # fit model
         model = fit_lstm(train, n_lag, n_seq, n_batch, n_epochs, n_neurons)
         # make forecasts
         forecasts = make_forecasts(model, n_batch, train, test, n_lag, n_seq)
         # inverse transform forecasts and test
         forecasts = inverse_transform(series, forecasts, scaler, n_test+n_seq-1)
         actual = [row[n_lag:] for row in test]
         actual = inverse_transform(series, actual, scaler, n_test+n_seq-1)
         # evaluate forecasts
         evaluate_forecasts(actual, forecasts, n_lag, n_seq)
         # plot forecasts
         #print(forecasts)

         plot_forecasts(series, forecasts, n_test+n_seq-1)
```

```
t+1 RMSE: 2.132839
t+2 RMSE: 2.145712
t+3 RMSE: 2.146527
t+4 RMSE: 2.172520
```



The plot above shows the comparison between the prediction values (red lines) and real data (blue lines) for all the time intervals. In order to take a look at the predition clearly, we plot only the last 100 time steps and the 4 time step prediction every 4 time step as below:

```
In [32]: n_plot=100
         forep=forecasts[-n_plot-1:-1]
         plot_forecasts(series[-110:-1], forep, n_plot+n_seq-1)
```



The prediction can almost predict the mean level of the login count in the next 4 time intervals (1 hour). The RMSE for the 4 intervals are 2.133, 2.146, 2.147, 2.173 respectively. This accuracy is good for busy time period, which usually has average login count of about 8 and the percentage error is about 25%. However, for the other time period with average login count below 4 or even equals 0, the percentage error becomes 50% to over 100%. This significant error is usually difficult to avoid. If we choose Mean Absolute Percentage Error (MAPE) as the loss function in the model, this percentage error may decrease for time period with low mean values but may increase the percentage error for the busy time period. We need to make a trade off according to which time period is most important for our model. Another reason that may enlarge the error is that the data for training is the fisrt 6-month data while the testing data is the last 2-month data, which means the model may perform well for the first 6-month training data but not as well for the testing data. The problem can be solved by adding the newest data into the training data set and updating the model as time goes on.

Besides the model using 1-day data as input (n_lag=96) described above, we also tried two other models using 1-hour and 1-week data as input (n_lag=4 and 672). Both of the two new models have larger RMSE (not shown in this document). Thus the best length of input time interval must be between 1 hour and 1 week and more tests are needed to find it.

Due to the limit of time, processor speed and memory size, the above model is only an initial result, which is obtained after only several tests of different training parameters. Change the numbers of training epoch, neuron, hidden layer, input time intervals as well as other parameters and methods must result in better prediction result and need to be tried in the future.

1.3

The probability distribution of the stochastic values should obey a normal distribution $n(\mu, \sigma^2)$ with mean value μ and variance σ^2. Thus, we can make a prediction interval with a given confident level (95% for example), which can be write as $\hat{y} = \hat{\mu} \pm z_\alpha \hat{\sigma}$, where $z_\alpha = 1.96$ for 95% confidence. The prediction of mean value $\hat{\mu}$ is the output value of the model and the square root of variance $\hat{\sigma}$ can be estimated be several method. The simplest one this using the mean values of the residuals e (difference between the prediction value and the real value). The RMSE is also a good estimator of the variance. Another method is the bootstrap residuals. In this method, we randomly select the residuals of different data point in to a residuals sample with large enough size and using the lower and upper bound percentiles to obtain a bootstrap interval ($[e_{0.025}, e_{0.975}]$ for 95% confident level), which can be applied to the prediction value to make a final confident interval of the prediction.

2 Part 2: Programming Exercise

In this problem, we first set a dictionary variable 'dict' to store the labels of cuisines and the corresponding numbers of dishes for each cuisine. The indexes are the labels of cuisines (A,B,C..), following the indexes are lists of integers that storing all possible numbers of dishes for the corresponding cuisines. We use list of integers instead of string here to avoid the possible problem caused by that the number of dishes may be larger than 10.

We write a function 'get_menus' to solve this problem. This function searches all possible combinations of cuisines and dishes. The input is the dictionary of cuisines and dishes we defined

and the output this the menu with all cuisine dish combinations. The code of this function is:

```
In [8]: def get_menus(dict):
                menus = []
                for cuisine, dishes in dict.items():
                    if not dishes == []: #exclude cuisines with no available dishes
                        menus = [menu + cuisine + str(dish) for dish in dishes \
                        for menu in menus or ['']]
                return menus
```

Below are several test cases, which include both cases with dish number larger than 10 and no available dish. The outputs are all correct.

```
In [10]: dict = {"A":[1], "B":[2,3], "C":[5,7,9]}
         result = get_menus(dict)
         print(result)
```

```
['A1B2C5', 'A1B3C5', 'A1B2C7', 'A1B3C7', 'A1B2C9', 'A1B3C9']
```

```
In [11]: dict = {"A":[7,4], "B":[2,11], "C":[3,8,5]}
         result = get_menus(dict)
         print(result)
```

```
['A7B2C3', 'A4B2C3', 'A7B11C3', 'A4B11C3', 'A7B2C8', 'A4B2C8', 'A7B11C8', 'A4B11C8',
'A7B2C5', 'A4B2C5', 'A7B11C5', 'A4B11C5']
```

```
In [13]: dict = {"A":[7,4], "B":[2,5], "C":[], "E":[2,5,8]}
         result = get_menus(dict)
         print(result)
```

```
['A7B2E2', 'A4B2E2', 'A7B5E2', 'A4B5E2', 'A7B2E5', 'A4B2E5', 'A7B5E5', 'A4B5E5',
'A7B2E8', 'A4B2E8', 'A7B5E8', 'A4B5E8']
```

3 Appendix

The codes of functions used in building the prediction model are listed below:

```
In [28]: # convert time series into supervised learning problem
         def series_to_supervised(data, n_in=1, n_out=1, dropnan=True):
             n_vars = 1 if type(data) is list else data.shape[1]
             df = DataFrame(data)
             cols, names = list(), list()
             # input sequence (t-n, ... t-1)
             for i in range(n_in, 0, -1):
                 cols.append(df.shift(i))
                 names += [('var%d(t-%d)' % (j+1, i)) for j in range(n_vars)]
```

```
    # forecast sequence (t, t+1, ... t+n)
    for i in range(0, n_out):
        cols.append(df.shift(-i))
        if i == 0:
            names += [('var%d(t)' % (j+1)) for j in range(n_vars)]
        else:
            names += [('var%d(t+%d)' % (j+1, i)) for j in range(n_vars)]
    # put it all together
    agg = concat(cols, axis=1)
    agg.columns = names
    # drop rows with NaN values
    if dropnan:
        agg.dropna(inplace=True)
    return agg


# create a differenced series
def difference(dataset, interval=1):
    diff = list()
    for i in range(interval, len(dataset)):
        value = dataset[i] - dataset[i - interval]
        diff.append(value)
    return Series(diff)


# transform series into train and test sets for supervised learning
def prepare_data(series, n_test, n_lag, n_seq):
    # extract raw values
    raw_values = series.values
    # transform data to be stationary
    #diff_series = difference(raw_values, 1)
    #diff_values = diff_series.values
    #diff_values = diff_values.reshape(len(diff_values), 1)
    diff_values=raw_values.reshape(len(raw_values), 1)
    # rescale values to -1, 1
    scaler = MinMaxScaler(feature_range=(-1, 1))
    scaled_values = scaler.fit_transform(diff_values)
    scaled_values = scaled_values.reshape(len(scaled_values), 1)

    # transform into supervised learning problem X, y
    supervised = series_to_supervised(scaled_values, n_lag, n_seq)
    supervised_values = supervised.values
    # split into train and test sets
    train, test = supervised_values[0:-n_test], supervised_values[-n_test:]
    return scaler, train, test


# fit an LSTM network to training data
```

```
def fit_lstm(train, n_lag, n_seq, n_batch, nb_epoch, n_neurons):
    # reshape training into [samples, timesteps, features]
    X, y = train[:, 0:n_lag], train[:, n_lag:]
    X = X.reshape(X.shape[0], 1, X.shape[1])
    # design network
    model = Sequential()
    model.add(LSTM(n_neurons, batch_input_shape=(n_batch, X.shape[1], X.shape[2]), \
    return_sequences=True))
    model.add(Dropout(0.2))
    model.add(LSTM(n_neurons, return_sequences=False))
    model.add(Dropout(0.2))
    model.add(Dense(y.shape[1]))
    model.compile(loss='mean_squared_error', optimizer='adam')
    # fit network
    for i in range(nb_epoch):
        #print(i)
        model.fit(X, y, epochs=1, batch_size=n_batch, verbose=0, shuffle=False)
        model.reset_states()
    return model


# make one forecast with an LSTM,
def forecast_lstm(model, X, n_batch):
    # reshape input pattern to [samples, timesteps, features]
    X = X.reshape(1, 1, len(X))
    # make forecast
    forecast = model.predict(X, batch_size=n_batch)
    # convert to array
    return [x for x in forecast[0, :]]


# evaluate the persistence model
def make_forecasts(model, n_batch, train, test, n_lag, n_seq):
    forecasts = list()
    for i in range(len(test)):
        X, y = test[i, 0:n_lag], test[i, n_lag:]
        # make forecast
        forecast = forecast_lstm(model, X, n_batch)
        # store the forecast
        forecasts.append(forecast)
    return forecasts


# invert differenced forecast
def inverse_difference(last_ob, forecast):
    # invert first forecast
    inverted = list()
    #inverted.append(forecast[0] + last_ob)
```

```
    inverted.append(forecast[0])
    # propagate difference forecast using inverted first value
    for i in range(1, len(forecast)):
        #inverted.append(forecast[i] + inverted[i-1])
        inverted.append(forecast[i])
    return inverted


def inverse_transform(series, forecasts, scaler, n_test):
    inverted = list()
    for i in range(len(forecasts)):
        # create array from forecast
        forecast = array(forecasts[i])
        forecast = forecast.reshape(1, len(forecast))
        # invert scaling
        inv_scale = scaler.inverse_transform(forecast)
        inv_scale = inv_scale[0, :]
        # invert differencing
        index = len(series) - n_test + i - 1
        last_ob = series.values[index]
        inv_diff = inverse_difference(last_ob, inv_scale)
        #inv_diff = inv_scale
        # store
        inverted.append(inv_diff)
    return inverted


# evaluate the RMSE for each forecast time step
def evaluate_forecasts(test, forecasts, n_lag, n_seq):
    for i in range(n_seq):
        actual = [row[i] for row in test]
        predicted = [forecast[i] for forecast in forecasts]
        rmse = sqrt(mean_squared_error(actual, predicted))
        print('t+%d RMSE: %f' % ((i+1), rmse))

# plot the forecasts in the context of the original dataset
def plot_forecasts(series, forecasts, n_test):
    # plot the entire dataset in blue
    pyplot.plot(series.values,'b-')
    # plot the forecasts in red
    for i in range(len(forecasts)-1,0,-4):
        off_s = len(series) - n_test + i - 1
        off_e = off_s + len(forecasts[i]) + 1
        xaxis = [x for x in range(off_s, off_e)]
        yaxis = [series.values[off_s]] + forecasts[i]
        pyplot.plot(xaxis, yaxis, 'r')
    # show the plot
    pyplot.show()
```