# Notebook to accompany NITC Final Project Report 1299

This notebook can be run to replicate the results described in NITC Final Project Report 1299. It is self-contained with the following caveat: if it is moved out of the Google Colab environment, then the libraries used may have to be installed by the user, e.g., `tensorflow`. If run within Colab, then a `Runtime/Run all` should run to completion.

```
import pandas as pd
import numpy as np
```

## I. Bring in the raw captured data

This is the data captured from the 18th and Alder intersection in Eugene OR during 12 days in June 2018. It has been pre-formatted into csv format.

```
raw_table = pd.read_csv('https://docs.google.com/spreadsheets/d/e/2PACX-1vTYrjPmo4rO3qoX3k5de
```

```
len(raw_table)
```

```
42920
```

```
raw_table.head()
```

| | date | time | x | y |
|---|---|---|---|---|
| **0** | 06/15/2018 | 2:04:43 | NaN | NaN |
| **1** | 06/15/2018 | 2:04:47 | 4.0 | NaN |
| **2** | 06/15/2018 | 2:04:52 | NaN | NaN |
| **3** | 06/15/2018 | 2:04:56 | 2.0 | 6.0 |
| **4** | 06/15/2018 | 2:05:05 | NaN | NaN |

```
raw_table.describe()
```

| | x | y |
|---|---|---|
| **count** | 22535.000000 | 12236.000000 |
| **mean** | 3.239627 | 6.446224 |
| **std** | 1.503878 | 0.832700 |
| **min** | 2.000000 | 6.000000 |
| **25%** | 2.000000 | 6.000000 |

# II. Wrangling

We will take a set of wrangling steps to get the data in a form that is usable for machine-learning.

## 1. Fill NaNs with 0

```
wrangled_table = raw_table.fillna(0)
wrangled_table.head()
```

| | date | time | x | y |
|---|---|---|---|---|
| **0** | 06/15/2018 | 2:04:43 | 0.0 | 0.0 |
| **1** | 06/15/2018 | 2:04:47 | 4.0 | 0.0 |
| **2** | 06/15/2018 | 2:04:52 | 0.0 | 0.0 |
| **3** | 06/15/2018 | 2:04:56 | 2.0 | 6.0 |
| **4** | 06/15/2018 | 2:05:05 | 0.0 | 0.0 |

## 2. Drop yellows

```
wrangled_table = wrangled_table[(wrangled_table.x != 0.) | (wrangled_table.y != 0.)].reset_in
```

```
wrangled_table.head()
```

| | date | time | x | y |
|---|---|---|---|---|

## 3. Create new `phase` column

This column will be binary.

```
      3  06/15/2018  2:05:20  2.0  6.0
```

```
phase_mapping = {(2,6): 1,  #bike friendly
                 (2,0): 1,  #bike friendly
                 (6,0): 1,  #bike friendly
                 (3,0): 0,
                 (4,0): 0,
                 (8,0): 0,
                 (3,8): 0,
                 (4,8): 0}
wrangled_table['phase'] = wrangled_table.apply(lambda row: phase_mapping[(row['x'], row['y'])
wrangled_table.head()
```

| | date | time | x | y | phase |
|---|---|---|---|---|---|
| **0** | 06/15/2018 | 2:04:47 | 4.0 | 0.0 | 0 |
| **1** | 06/15/2018 | 2:04:56 | 2.0 | 6.0 | 1 |
| **2** | 06/15/2018 | 2:05:10 | 4.0 | 0.0 | 0 |
| **3** | 06/15/2018 | 2:05:20 | 2.0 | 6.0 | 1 |
| **4** | 06/15/2018 | 2:05:28 | 4.0 | 8.0 | 0 |

```
wrangled_table.tail()
```

| | date | time | x | y | phase |
|---|---|---|---|---|---|
| **22530** | 06/27/2018 | 9:22:37 | 4.0 | 8.0 | 0 |
| **22531** | 06/27/2018 | 9:22:58 | 2.0 | 6.0 | 1 |
| **22532** | 06/27/2018 | 9:23:16 | 8.0 | 0.0 | 0 |
| **22533** | 06/27/2018 | 9:23:19 | 4.0 | 8.0 | 0 |
| **22534** | 06/27/2018 | 9:23:37 | 2.0 | 6.0 | 1 |

```
for i in range(2):
  print(i, wrangled_table.phase.to_list().count(i)/len(wrangled_table))  #class and percentag
```

```
0 0.5781229199023741
1 0.4218770800976259
```

58% of rows are 0 (not bike friendly)

## 4. Drop all other columns

```
wrangled_table = wrangled_table[['phase']]
```

```
len(wrangled_table)
```

```
22535
```

```
wrangled_table.head()
```

| | phase |
|---|---|
| **0** | 0 |
| **1** | 1 |
| **2** | 0 |
| **3** | 1 |
| **4** | 0 |

## 5. Convert column values to `float32`

```
wrangled_table = wrangled_table.astype({'phase': np.float32})
```

```
wrangled_table.head()
```

| | phase |
|---|---|
| **0** | 0.0 |
| **1** | 1.0 |
| **2** | 0.0 |
| **3** | 1.0 |
| **4** | 0.0 |

# III. Train/Test split

We choose to use an 80/20 split.

```
n = int(.8*len(wrangled_table))
training_table = wrangled_table[:n]
test_table = wrangled_table[n:]
```

# IV. Reformat the data as a time-series problem

First, we need to choose a lookback value. This is a hyperparameter that can range from 1 (lookback just one phase) to hundreds (lookback to last 100 phases). After exploration, we chose a value of 7.

```
n_input = 7  #lookback of 7
```

Next we define a function that will help us reconstitute our tables into a matrix form.

```
def split_sequences(sequences, n_steps):
  X, y = list(), list()
  for i in range(len(sequences)):
    # find the end of this pattern
    end_ix = i + n_steps
    # check if we are beyond the dataset
    if end_ix+1 > len(sequences):
      break
    # gather input and output parts of the pattern
    seq_x, seq_y = sequences[i:end_ix], sequences[end_ix]
    X += [seq_x]
    y += [seq_y]
  return np.array(X), np.array(y)
```

# 1. Convert the training table into a list of values

```
dataset = [x[0] for x in training_table.values.tolist()]
dataset[:5]
```

```
[0.0, 1.0, 0.0, 1.0, 0.0]
```

# 2. Rework the training data

In essence, we want (1) a `train_x` which is a list of rows where each row is 7 values, and (2) a `train_y` which are the labels for the rows where a label is the 8th value for the row.

```
# split into training pieces

train_x, train_y = split_sequences(dataset, n_input)
```

Here is an example of the first row and label.

```
print(train_x[10])
print(train_y[10])
```

```
     [1. 0. 1. 0. 1. 0. 1.]
     0.0
```

```
#reshape for tensorflow
train_x = train_x.reshape(train_x.shape[0], train_x.shape[1], 1)
```

## 3. Convert the test table into a list of values

```
dataset = [x[0] for x in test_table.values.tolist()]
dataset[:5]
```

```
     [0.0, 1.0, 0.0, 0.0, 1.0]
```

## 4. Rework the test data

In essence, we want (1) a `test_x` which is a list of rows where each row is 7 values, and (2) a `test_y` which are the labels for the rows where a label is the 8th value for the row.

```
test_x, test_y = split_sequences(dataset, n_input)

test_x = test_x.reshape(test_x.shape[0], test_x.shape[1], 1)
```

```
test_x.shape
```

```
     (4500, 7, 1)
```

```
test_y.shape
```

```
     (4500,)
```

# V. Time Series Forecasting With a Univariate CNN

## Step 1. Define metrics used

We have chosen to work with Precision and Recall.

```
def metrics(p_a_pairs):
  tn = p_a_pairs.count((0,0))
  tp = p_a_pairs.count((1,1))
  fp = p_a_pairs.count((1,0))
  fn = p_a_pairs.count((0,1))
  print(f'Confusion matrix: {[tp, fp, tn, fn]}')
  print(f'Precision:\t{(tp)/(tp+fp)}')
  print(f'Recall:\t{(tp)/(tp+fn)}')
  print(f'Accuracy:\t{(tp+tn)/len(p_a_pairs)}')
  return [tp/(tp+fp), tp/(tp+fn), (tp+tn)/len(p_a_pairs)]
```

## Step 2. Bring in Tensorflow 1D CNN pieces

```
import tensorflow
import tensorflow as tf
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import Dropout
from tensorflow.keras.layers import Flatten
from tensorflow.keras.layers import Conv1D
from tensorflow.keras.layers import MaxPooling1D
import matplotlib.pyplot as plt
```

## Step 3. Set seed so can replicate results

```
tensorflow.random.set_seed(1234)
```

## Step 4. Choose epochs

```
epochs = 5
```

## Step 5. Train

Roughly 3 minutes.

```
%%time
n_timesteps= train_x.shape[1]  #e.g., 7

# define model
c_model = Sequential()
c_model.add(Conv1D(filters=16, kernel_size=4, activation='selu', input_shape=(n_timesteps,1))
c_model.add(Dropout(0.4))
c_model.add(Conv1D(filters=8, kernel_size=3, activation='selu'))
c_model.add(Dropout(0.4))
c_model.add(Conv1D(filters=4, kernel_size=2, activation='selu'))
c_model.add(Dropout(0.4))
c_model.add(Flatten())
c_model.add(Dense(20, activation='selu'))
c_model.add(Dropout(0.2))
c_model.add(Dense(10, activation='selu'))
c_model.add(Dropout(0.2))
c_model.add(Dense(1, activation='sigmoid'))
c_model.compile(optimizer='adam',
            loss='binary_crossentropy',
            metrics=['accuracy'])

# fit network
training = c_model.fit(train_x, train_y, epochs=epochs, batch_size=1, verbose=1)

# plot training history
plt.plot(training.history['accuracy'])
plt.plot(training.history['loss'])
plt.title('model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['binary accuracy', 'loss'], loc='upper left')
plt.show()
```

```
Epoch 1/5
18021/18021 [==============================] - 35s 2ms/step - loss: 0.4157 - accuracy: 0
Epoch 2/5
18021/18021 [==============================] - 34s 2ms/step - loss: 0.3701 - accuracy: 0
Epoch 3/5
18021/18021 [==============================] - 35s 2ms/step - loss: 0.3731 - accuracy: 0
Epoch 4/5
18021/18021 [==============================] - 35s 2ms/step - loss: 0.3692 - accuracy: 0
Epoch 5/5
18021/18021 [==============================] - 35s 2ms/step - loss: 0.3700 - accuracy: 0
```




## Step 6. Produce raw output from test data

```
# calculate predictions
cnn_out = c_model.predict(test_x)  #nested format
cnn_raw = [x[0] for x in cnn_out]  #convert to straight list of numbers
```

```
cnn_raw[:10]
```

```
[0.0003219545,
 0.7129552,
 0.0002809763,
 0.7159964,
 0.00026601553,
 0.7159964,
 0.00026601553,
 0.7159964,
 0.7103143,
 0.00026017427]
```

## Step 7. Report results

Note using default threshold value of `.5`.

```
metrics(list(zip([1 if p>=.5 else 0 for p in cnn_raw], test_y)))
```

```
Confusion matrix: [1922, 654, 1922, 2]
Precision:      0.7461180124223602
Recall: 0.998960498960499
Accuracy:       0.8542222222222222
[0.7461180124223602, 0.998960498960499, 0.8542222222222222]
```

# VI. Time Series Forecasting With an LSTM

## Step 1. Bring in TensorFlow pieces needed

```
from tensorflow.keras.layers import LSTM
```

## Step 2. Define LSTM model

```
lstm_size = 50

ls_model = Sequential()
ls_model.add(LSTM(lstm_size, activation='selu', input_shape=(n_timesteps,1), return_sequences
ls_model.add(Dropout(0.2))
ls_model.add(LSTM(lstm_size, activation='selu'))
ls_model.add(Dropout(0.2))
ls_model.add(Dense(20, activation='selu'))
ls_model.add(Dropout(0.2))
ls_model.add(Dense(10, activation='selu'))
ls_model.add(Dropout(0.2))
ls_model.add(Dense(1, activation='sigmoid'))
ls_model.compile(optimizer='adam',
            loss='binary_crossentropy',
            metrics=['accuracy'])
```

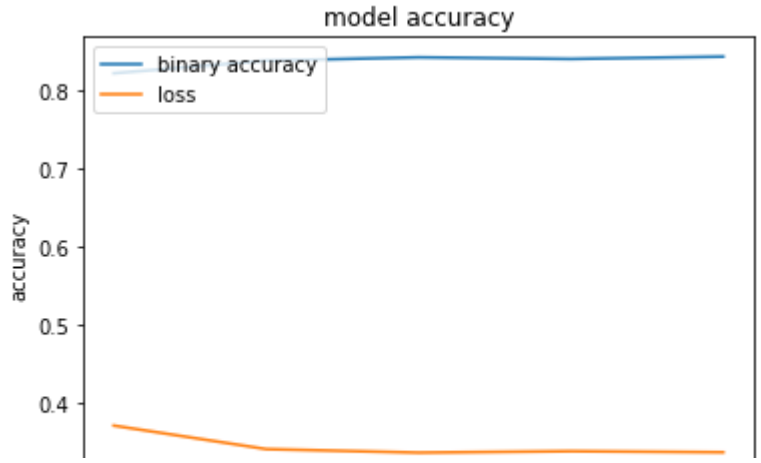
## Step 3. Train

Roughly 9 minutes.

```
%%time
training = ls_model.fit(train_x, train_y, epochs=epochs, batch_size=1, verbose=1)

plt.plot(training.history['accuracy'])
plt.plot(training.history['loss'])
plt.title('model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['binary accuracy', 'loss'], loc='upper left')
plt.show()
```

```
Epoch 1/5
18021/18021 [==============================] - 108s 6ms/step - loss: 0.3711 - accuracy:
Epoch 2/5
18021/18021 [==============================] - 107s 6ms/step - loss: 0.3410 - accuracy:
Epoch 3/5
18021/18021 [==============================] - 105s 6ms/step - loss: 0.3365 - accuracy:
Epoch 4/5
18021/18021 [==============================] - 104s 6ms/step - loss: 0.3383 - accuracy:
Epoch 5/5
18021/18021 [==============================] - 105s 6ms/step - loss: 0.3369 - accuracy:
```




## Step 4. Obtain raw predictions from test data

```
Wall time: 8min 50s
```

```
# calculate predictions
lstm_out = ls_model.predict(test_x)
lstm_raw = [x[0] for x in lstm_out]  #convert to straight list of numbers
```

```
lstm_raw[:10]
```

```
[0.00026986003,
 0.7747933,
 0.00038078427,
 0.7989631,
 0.00019875169,
 0.7989631,
 0.00019875169,
 0.7989631,
 0.7984611,
 1.9766556e-05]
```

## Step 5. Show results

```
metrics(list(zip([1 if p>=.5 else 0 for p in lstm_raw], test_y)))
```

```
Confusion matrix: [1881, 612, 1964, 43]
```

```
Precision:      0.7545126353790613
Recall: 0.9776507276507277
Accuracy:       0.8544444444444445
[0.7545126353790613, 0.9776507276507277, 0.8544444444444445]
```

# VII. Precision-Recall Tradeoff

We will contentrate on the raw output from the LSTM model.

## Step 1. Show in table form

```
lstm_table = pd.DataFrame(columns=['Threshold', 'Precision', 'Recall', 'Accuracy'])


thresholds = [.5, .55, .6, .65, .7, .71, .72, .73, .74, .75, .76, .77, .78, .79,.8, .85]


%%capture
for i,threshold in enumerate(thresholds):
  lstm_binary = [1 if r>=threshold else 0 for r in lstm_raw]  #decision rule
  p_a_pairs = list(zip(lstm_binary, test_y))
  row = [threshold]
  row += metrics(p_a_pairs)
  lstm_table.loc[i] = row


lstm_table
```

| | Threshold | Precision | Recall | Accuracy |
|---|---|---|---|---|
| **0** | 0.50 | 0.754513 | 0.977651 | 0.854444 |
| **1** | 0.55 | 0.764456 | 0.934511 | 0.848889 |
| **2** | 0.60 | 0.764781 | 0.934511 | 0.849111 |
| **3** | 0.65 | 0.772788 | 0.903326 | 0.845111 |

## Step 2. Show in graph form

```
from sklearn.metrics import precision_score, recall_score

thresholds = np.array(thresholds)
precision_scores = []
recall_scores = []
for threshold in thresholds:
    adjusted_predictions = [1 if p >= threshold else 0 for p in lstm_raw]
    precision_scores.append(precision_score(test_y, adjusted_predictions))
    recall_scores.append(recall_score(test_y, adjusted_predictions))
plt.plot(thresholds, precision_scores, label="precision")
plt.plot(thresholds, recall_scores, label="recall")
plt.legend()
plt.show()
```