Predictive Analytics

-Bosch Production Line Performance example
-Sine Wave prediction using LSTM

Overview

Topics include:

- Pre-requisites
- Introduction to Pandas, Numpy, Matplotlib
- About the Dataset and features
- Data Preprocessing
- Build the Model and Train
- Inference
- Conclusion

Pre-requisites

Option 1: Using DevCloud

- Access to DevCloud
- Jupyter Notebook with Anaconda for Python Libraries

Option 2: Running Locally

- Jupyter Notebook with Anaconda for Python Libraries

Toolset for DevCloud:

Intel® Distribution for Python

Comes with

- Accelerated performance from Intel's Math Kernel Library (MKL)
- Also contains Data Analytics Acceleration Library (DAAL), Message Passing Interface (MPI), and Threading Building Blocks (TBB)

Toolset:

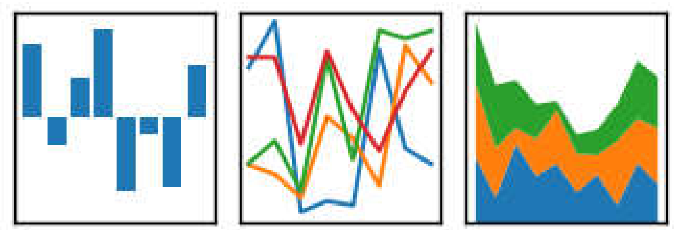
- **Jupyter notebooks:** interactive coding and visualization of output
- **NumPy, SciPy, Pandas:** numerical computation
- **Matplotlib, Seaborn:** data visualization
- **Scikit-learn:** machine learning

Introduction to Pandas

- Library for computation with tabular data
- Mixed types of data allowed in a single table
- Columns and rows of data can be named
- Advanced data aggregation and statistical functions

pandas

$$y_{it} = \beta' x_{it} + \mu_i + \epsilon_{it}$$



Source: http://pandas.pydata.org/

Introduction to Pandas

Basic data structures

Type	Pandas Name
Vector (1 Dimension)	Series
Array (2 Dimensions)	DataFrame

Pandas Series Creation and Indexing

Use data from step tracking application to create a Pandas Series

Code

```
import pandas as pd

step_data = [3620, 7891, 9761,
             3907, 4338, 5373]

step_counts = pd.Series(step_data,
                        name='steps')

print(step_counts)
```

Output

```
>>> 0 3620
    1 7891
    2 9761
    3 3907
    4 4338
    5 5373
    Name: steps, dtype: int64
```

Pandas Series Creation and Indexing

Add a date range to the Series

Code

```
step_counts.index = pd.date_range('20150329',
                                  periods=6)

print(step_counts)
```

Output

```
>>> 2015-03-29 3620
    2015-03-30 7891
    2015-03-31 9761
    2015-04-01 3907
    2015-04-02 4338
    2015-04-03 5373
    Freq: D, Name: steps,
    dtype: int64
```

Pandas Series Creation and Indexing

Select data by the index values

Code

```
# Just like a dictionary
print(step_counts['2015-04-01'])

# Or by index position--like an array
print(step_counts[3])

# Select all of April
print(step_counts['2015-04'])
```

Output

```
>>>  3907

>>> 3907

>>>  2015-04-01 3907
     2015-04-02 4338
     2015-04-03 5373
     Freq: D, Name: steps,
     dtype: int64
```

Pandas Data Types and Imputation

Data types can be viewed and converted

Code

```
# View the data type
print(step_counts.dtypes)

# Convert to a float
step_counts = step_counts.astype(np.float)

# View the data type
print(step_counts.dtypes)
```

Output

```
>>> int64

>>> float64
```

Pandas Data Types and Imputation

Invalid data points can be easily filled with values

Code

```
# Create invalid data
step_counts[1:3] = np.NaN

# Now fill it in with zeros
step_counts = step_counts.fillna(0.)
# equivalently,
# step_counts.fillna(0., inplace=True)

print(step_counts[1:3])
```

Output

```
>>> 2015-03-30 0.0
    2015-03-31 0.0
    Freq: D, Name: steps,
    dtype: float64
```

Pandas DataFrame Creation and Methods

DataFrames can be created from lists, dictionaries, and Pandas Series

Code

```
# Cycling distance
cycling_data = [10.7, 0, None, 2.4, 15.3,
                10.9, 0, None]

# Create a tuple of data
joined_data = list(zip(step_data,
                       cycling_data))

# The dataframe
activity_df = pd.DataFrame(joined_data)

print(activity_df)
```

Output

>>>

	0	1
0	3620	10.7
1	7891	0.0
2	9761	NaN
3	3907	2.4
4	4338	15.3
5	5373	10.9

Pandas DataFrame Creation and Methods

Labeled columns and an index can be added

Code

```
# Add column names to dataframe
activity_df = pd.DataFrame(joined_data,
              index=pd.date_range('20150329',
              periods=6),
              columns=['Walking','Cycling'])

print(activity_df)
```

Output

>>>

	Walking	Cycling
2015-03-29	3620	10.7
2015-03-30	7891	0.0
2015-03-31	9761	NaN
2015-04-01	3907	2.4
2015-04-02	4338	15.3
2015-04-03	5373	10.9

Indexing DataFrame Rows

DataFrame rows can be indexed by row using the 'loc' and 'iloc' methods

Code

```
# Select row of data by index name
print(activity_df.loc['2015-04-01'])
```

Output

```
>>> Walking 3907.0
    Cycling 2.4
    Name: 2015-04-01,
    dtype: float64
```

Indexing DataFrame Rows

DataFrame rows can be indexed by row using the 'loc' and 'iloc' methods

Code

```
# Select row of data by integer position
print(activity_df.iloc[-3])
```

Output

```
>>> Walking 3907.0
    Cycling 2.4
    Name: 2015-04-01,
    dtype: float64
```

Indexing DataFrame Columns

DataFrame columns can be indexed by name

Code

```
# Name of column
print(activity_df['Walking'])
```

Output

```
>>> 2015-03-29 3620
    2015-03-30 7891
    2015-03-31 9761
    2015-04-01 3907
    2015-04-02 4338
    2015-04-03 5373
    Freq: D, Name: Walking,
    dtype: int64
```

Indexing DataFrame Columns

DataFrame columns can also be indexed as properties

Code

```
# Object-oriented approach
print(activity_df.Walking)
```

Output

```
>>> 2015-03-29 3620
    2015-03-30 7891
    2015-03-31 9761
    2015-04-01 3907
    2015-04-02 4338
    2015-04-03 5373
    Freq: D, Name: Walking,
    dtype: int64
```

Indexing DataFrame Columns

DataFrame columns can be indexed by integer

Code

```
# First column
print(activity_df.iloc[:,0])
```

Output

```
>>> 2015-03-29 3620
    2015-03-30 7891
    2015-03-31 9761
    2015-04-01 3907
    2015-04-02 4338
    2015-04-03 5373
    Freq: D, Name: Walking,
    dtype: int64
```

Reading Data with Pandas

CSV and other common filetypes can be read with a single command

Code

```python
# The location of the data file
filepath = 'data/Iris_Data/Iris_Data.csv'

# Import the data
data = pd.read_csv(filepath)

# Print a few rows
print(data.iloc[:5])
```

Output

>>>

	sepal_length	sepal_width	petal_length	petal_width	species
0	5.1	3.5	1.4	0.2	Iris-setosa
1	4.9	3.0	1.4	0.2	Iris-setosa
2	4.7	3.2	1.3	0.2	Iris-setosa
3	4.6	3.1	1.5	0.2	Iris-setosa
4	5.0	3.6	1.4	0.2	Iris-setosa

Assigning New Data to a DataFrame

Data can be (re-)assigned to a DataFrame column

Code

```
# Create a new column that is a product
# of both measurements
data['sepal_area'] = data.sepal_length *
                     data.sepal_width

# Print a few rows and columns
print(data.iloc[:5, -3:])
```

Output

>>>

	petal_width	species	sepal_area
0	0.2	Iris-setosa	17.85
1	0.2	Iris-setosa	14.70
2	0.2	Iris-setosa	15.04
3	0.2	Iris-setosa	14.26
4	0.2	Iris-setosa	18.00

Applying a Function to a DataFrame Column

Functions can be applied to columns or rows of a DataFrame or Series

Code

```
# The lambda function applies what
# follows it to each row of data
data['abbrev'] = (data
                  .species
                  .apply(lambda x:
                   x.replace('Iris-','')))

# Note that there are other ways to
# accomplish the above

print(data.iloc[:5, -3:])
```

Output

>>>

	petal_width	species	abbrev
0	0.2	Iris-setosa	setosa
1	0.2	Iris-setosa	setosa
2	0.2	Iris-setosa	setosa
3	0.2	Iris-setosa	setosa
4	0.2	Iris-setosa	setosa

Concatenating Two DataFrames

Two DataFrames can be concatenated along either dimension

Code

```
# Concatenate the first two and
# last two rows
small_data = pd.concat([data.iloc[:2],
                        data.iloc[-2:]])


print(small_data.iloc[:,-3:])

# See the 'join' method for
# SQL style joining of dataframes
```

Output

>>>

	petal_length	petal_width	species
0	1.4	0.2	Iris-setosa
1	1.4	0.2	Iris-setosa
148	5.4	2.3	Iris-virginica
149	5.1	1.8	Iris-virginica

Aggregated Statistics with GroupBy

Using the groupby method calculated aggregated DataFrame statistics

Code

```
# Use the size method with a
# DataFrame to get count
# For a Series, use the .value_counts
# method
group_sizes = (data
               .groupby('species')
               .size())

print(group_sizes)
```

Output

```
>>> species
    Iris-setosa         50
    Iris-versicolor     50
    Iris-virginica      50
    dtype: int64
```

Performing Statistical Calculations

Pandas contains a variety of statistical methods—mean, median, and mode

Code	Output

```
# Mean calculated on a DataFrame
print(data.mean())
```

```
>>> sepal_length 5.843333
    sepal_width 3.054000
    petal_length 3.758667
    petal_width 1.198667
    dtype: float64
```

```
# Median calculated on a Series
print(data.petal_length.median())
```

```
>>> 4.35
```

```
# Mode calculated on a Series
print(data.petal_length.mode())
```

```
>>> 0 1.5
    dtype: float64
```

Performing Statistical Calculations

Standard deviation, variance, SEM and quantiles can also be calculated

Code

```
# Standard dev, variance, and SEM
print(data.petal_length.std(),
      data.petal_length.var(),
      data.petal_length.sem())
```

Output

Performing Statistical Calculations

Standard deviation, variance, SEM and quantiles can also be calculated

Code

```
# Standard dev, variance, and SEM
print(data.petal_length.std(),
      data.petal_length.var(),
      data.petal_length.sem())

# As well as quantiles
print(data.quantile(0))
```

Output

```
>>> 1.76442041995
    3.11317941834
    0.144064324021

>>> sepal_length 4.3
    sepal_width 2.0
    petal_length 1.0
    petal_width 0.1
    Name: 0, dtype: float64
```

Performing Statistical Calculations

Multiple calculations can be presented in a DataFrame

Code

```
print(data.describe())
```

Output

\>>>

	sepal_length	sepal_width	petal_length	petal_width
count	150.000000	150.000000	150.000000	150.000000
mean	5.843333	3.054000	3.758667	1.198667
std	0.828066	0.433594	1.764420	0.763161
min	4.300000	2.000000	1.000000	0.100000
25%	5.100000	2.800000	1.600000	0.300000
50%	5.800000	3.000000	4.350000	1.300000
75%	6.400000	3.300000	5.100000	1.800000
max	7.900000	4.400000	6.900000	2.500000

Sampling from DataFrames

DataFrames can be randomly sampled from

Code

```
# Sample 5 rows without replacement
sample = (data
          .sample(n=5,
                  replace=False,
                  random_state=42))


print(sample.iloc[:,-3:])
```

Output

>>>

	petal_length	petal_width	species
73	4.7	1.2	Iris-versicolor
18	1.7	0.3	Iris-setosa
118	6.9	2.3	Iris-virginica
78	4.5	1.5	Iris-versicolor
76	4.8	1.4	Iris-versicolor

SciPy and NumPy also contain a variety of statistical functions.

Introduction to Numpy

- Library for manipulating Large arrays and matrices of numeric data
- Functions available to perform standard vector and matrix multiplication
- Methods for working with polynomials and derivatives
- Provides routines for discrete fourier transformation and more complex linear algebra operations

Source: https://docs.scipy.org/doc/numpy/

Numpy Arrays - Basics

Every element in a Numpy array must be of the same type

Code

```
import numpy as np

a = np.array([1, 4, 5, 8], float)
a
# Multidimensional arrays
a = np.array([[1, 2, 3], [4, 5, 6]], float)
a
# slicing the array
a[1,:]
a.Shape
a.Dtype
len(a)  (returns the length of first axis)
```

Output

```
>>> array([ 1., 4., 5., 8.])

>>> array([[ 1., 2., 3.],
           [ 4., 5., 6.]])
>>> array([ 4., 5., 6.])
>>> (2,3)
>>> dtype('float64')
>>> 2
```

Numpy Arrays - Basics

Code

```
# in statement used to test values present
in an array
2 in a
# reshaping arrays
a = np.array(range(10), float)
a
a = a.reshape((5, 2))
a
#Other operations
tolist()  -- Create list from arrays
tostring() – raw data array to binary string
```

Output

```
>>> True
>>> array([ 0., 1., 2., 3.,
4., 5., 6., 7., 8., 9.])
>>> array([[ 0., 1.],
           [ 2., 3.],
           [ 4., 5.],
           [ 6., 7.],
           [ 8., 9.]])
```

Numpy Arrays - Operations

Filling, flatten, transpose and concatenate operations on arrays

Code

```
a = array([1, 2, 3], float)
a.fill(0)

a = np.array([[1, 2, 3], [4, 5, 6]], float)
a.flatten()
a.transpose()

a = np.array([1,2], float)
b = np.array([3,4,5,6], float)
c = np.array([7,8,9], float)
np.concatenate((a, b, c))
```

Output

```
>>> array([ 1., 2., 3.])
>>> array([[ 0., 0., 0.])
>>> array([[ 1., 2., 3.],
           [ 4., 5., 6.]])
>>> array([ 1., 2., 3., 4.,
            5., 6.])
>>> array([[ 1., 4.],
           [ 2., 5.],
           [ 3., 6.]])
array([1., 2., 3., 4., 5., 6.,
7., 8., 9.])
```

Numpy Arrays – Array Mathematics

Standard mathematical operations are applied on an element by element basis on arrays

Code

```
a = np.array([1,2,3], float)
b = np.array([5,2,6], float)
a + b
a – b
a * b
b / a
a % b
b**a
a = np.zeros((2,2), float)
# other mathematical functions
Abs(), sign(), sqrt(), log(), log10(),
exp(), sin(), cos(), tan(), arcsin(),
arcos(), arctan(), sinh(), cosh(), tanh(),
arcsinh(), arccosh(), and arctanh(),
floor(), ceil(), rint(), sum(), prod()
```

Output

```
>>>  array([6., 4., 9.])
>>>  array([-4., 0., -3.])
>>> array([5., 4., 18.])
>>> array([5., 1., 2.])
>>> array([1., 0., 3.])
>>> array([5., 4., 216.])
>>> array([[ 0., 0.],
           [ 0., 0.]])
```

intel Software

Numpy Arrays – Array Mathematics

Extracting whole-array properties

Code	Output
`a = np.array([2, 4, 3], float)`	
`a.sum()`	`>>> 9`
`a.prod()`	`>>> 24`
`a.mean()`	`>>> 3`
`a.var()`	`>>> 0.6666`
`a.std()`	`>>> 0.8164`
`a.argmin()`	`>>> 2`
`a.argmax()`	`>>> 4`
`a.sort()`	`>>> array([2,3,4])`
`a = np.array([1, 1, 4, 5, 5, 5, 7], float)`	
`a.unique()`	`>>> array([ 1., 4., 5., 7.])`
`a = np.array([[1, 2], [3, 4]], float)`	
`a.diagonal()`	`>>> array([ 1., 4.])`

Numpy Arrays – Vector and Matrix mathematics

Functions for Vector and Matrix multiplications

Code

```
a = np.array([1, 2, 3], float)
b = np.array([0, 1, 1], float)
np.dot(a, b)
a = np.array([[0, 1], [2, 3]], float)
b = np.array([2, 3], float)
c = np.array([[1, 1], [4, 0]], float)
np.dot(b, a)
np.dot(a, b)
np.dot(a, c)
np.dot(c, a)
#Numpy comes with many built in routines for
linear algebra calculations and statistics
```

Output

```
>>>  5.0

>>> array([ 6., 11.])
>>> array([ 3., 13.])
>>> array([[ 4., 0.],
           [ 14., 2.]])
>>> array([[ 2., 4.],
           [ 0., 4.]])
```

Visualization Libraries

Visualizations can be created in multiple ways:

- Matplotlib
- Pandas (via Matplotlib)
- Seaborn
  - Statistically-focused plotting methods
  - Global preferences incorporated by Matplotlib

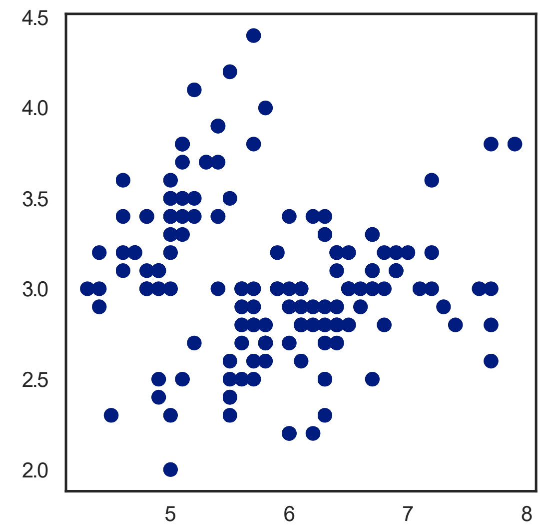
Basic Scatter Plots with Matplotlib

Scatter plots can be created from Pandas Series

Code

```
Import matplotlib.pyplot as plt


plt.plot(data.sepal_length,
         data.sepal_width,
         ls ='', marker='o')
```

Output



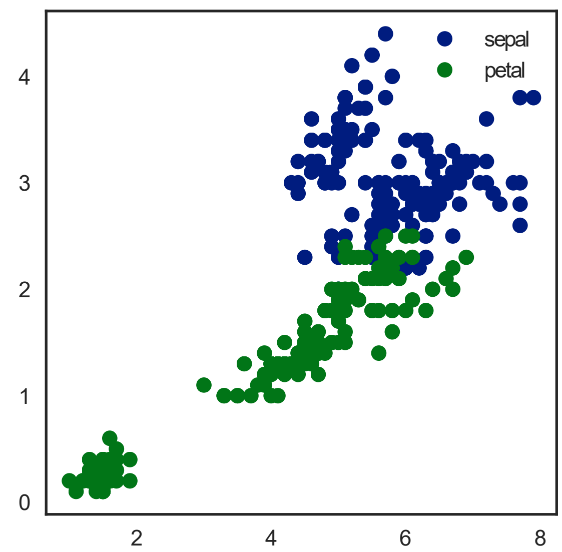
Basic Scatter Plots with Matplotlib

Multiple layers of data can also be added

Code

```
plt.plot(data.sepal_length,
         data.sepal_width,
         ls ='', marker='o',
         label='sepal')

plt.plot(data.petal_length,
         data.petal_width,
          ls ='', marker='o',
         label='petal')
```

Output



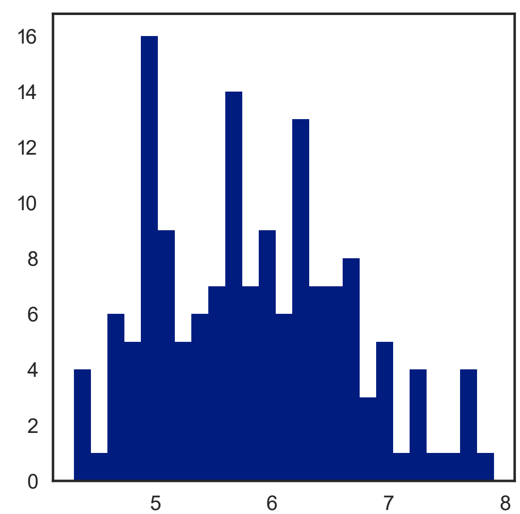
Histograms with Matplotlib

Histograms can be created from Pandas Series

Code

```
plt.hist(data.sepal_length, bins=25)
```

Output



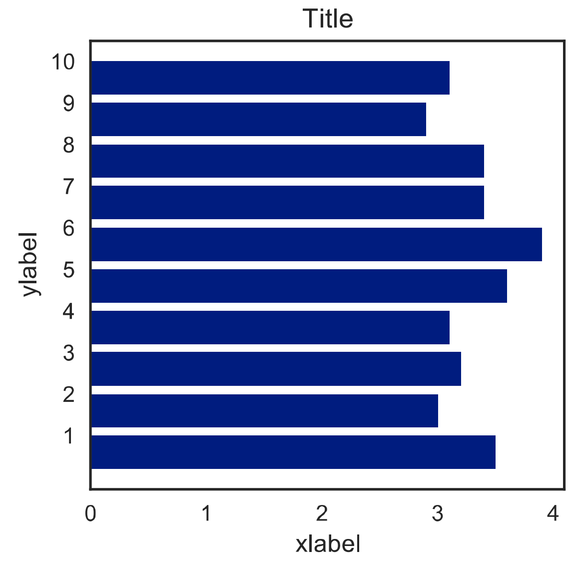
Customizing Matplotlib Plots

Every feature of Matplotlib plots can be customized

Code

```
fig, ax = plt.subplots()

ax.barh(np.arange(10),
        data.sepal_width.iloc[:10])

# Set position of ticks and tick labels
ax.set_yticks(np.arange(0.4,10.4,1.0))
ax.set_yticklabels(np.arange(1,11))
ax.set(xlabel='xlabel', ylabel='ylabel',
       title='Title')
```

Output



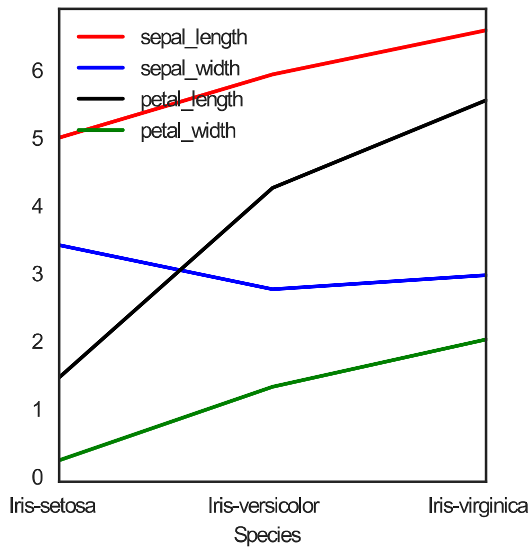
Incorporating Statistical Calculations

Statistical calculations can be included with Pandas methods

Code

```
(data
 .groupby('species')
 .mean()
 .plot(color=['red','blue',
              'black','green'],
       fontsize=10.0, figsize=(4,4)))
```

Output



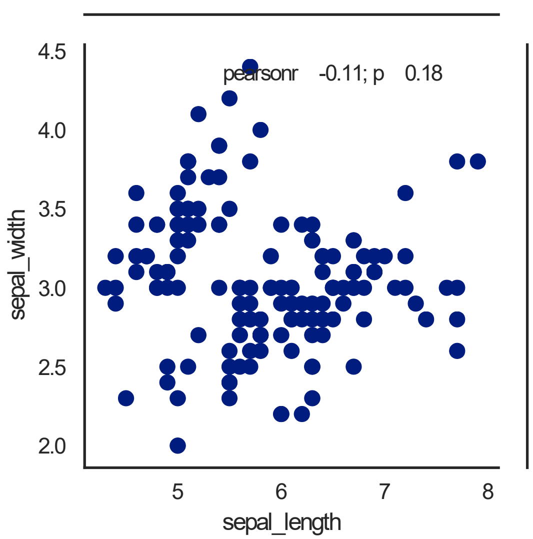
Statistical Plotting with Seaborn

Joint distribution and scatter plots can be created

Code

```
import seaborn as sns

sns.jointplot(x='sepal_length',
              y='sepal_width',
              data=data, size=4)
```

Output



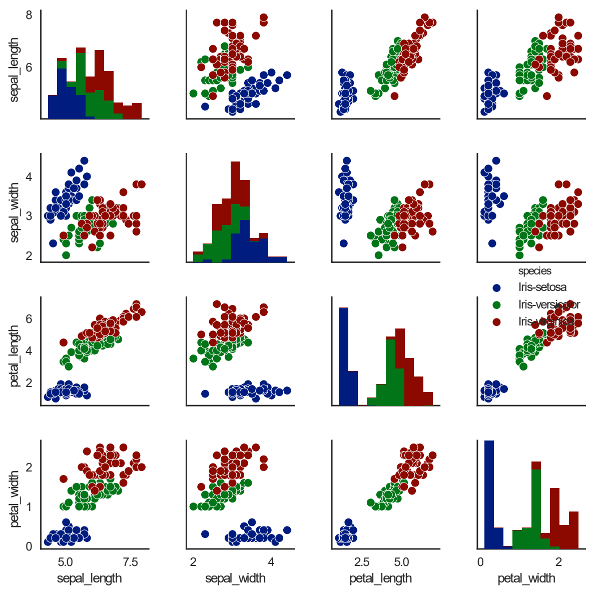
Statistical Plotting with Seaborn

Correlation plots of all variable pairs can also be made with Seaborn

Code

```
sns.pairplot(data, hue='species', size=3)
```

Output



About Bosch Dataset

- Represents measurements of parts moving through production lines
- Each part has a unique Id. The Response variable value decides the quality control outcome of the part
- The data consists of large number of anonymized features
- Features represented as Lxx_Sxxx_Fxxxx
- E.g. L3_S50_F4245. Feature number 4245 measured in line 3, station 50
- Data is organized into separate files by feature type – numerical, categorical and date
- Date feature provide timestamp when the feature was taken – viz, L0_S0_D1 is the time when the L0_S0_F0 was taken

The data is organized into the following files for train and test:

- train_numeric.csv & test_numeric.csv - the training and test set numeric features
- train_categorical.csv & test_categorical.csv - the training and test set categorical features
- train_date.csv & test _date.csv - the training and test set date features

Reading Data with Pandas

Reading Numeric data

Code

```
# The location of the data file
filepath =
'~/data/bosch_data/train_numeric.csv'

# Import the data
df_numeric = pd.read_csv(filepath)

# Print a few rows
print(df_numeric.head())
```

Output

>>>

	Id	L0_S0_F0	L0_S0_F2	L0_S0_F4	L0_S0_F6	L0_S0_F8	L0_S0_F10	L0_S0_F12	L0_S0_F14	L0_S0_F16	...
0	4	0.030	-0.034	-0.197	-0.179	0.118	0.116	-0.015	-0.032	0.020	...
1	6	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...
2	7	0.088	0.086	0.003	-0.052	0.161	0.025	-0.015	-0.072	-0.225	...
3	9	-0.036	-0.064	0.294	0.330	0.074	0.161	0.022	0.128	-0.026	...
4	11	-0.055	-0.086	0.294	0.330	0.118	0.025	0.030	0.168	-0.169	...

Reading Data with Pandas

Reading date data

Code

```
# The location of the data file
filepath =
'~/data/bosch_data/train_date.csv'

# Import the data
df_date = pd.read_csv(filepath)

# Print a few rows
print(df_date.head(10))
```

Output

>>>

	Id	L0_S0_D1	L0_S0_D3	L0_S0_D5	L0_S0_D7	L0_S0_D9	L0_S0_D11	L0_S0_D13	L0_S0_D15	L0_S0_D17	...
0	4	82.24	82.24	82.24	82.24	82.24	82.24	82.24	82.24	82.24	...
1	6	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...
2	7	1618.70	1618.70	1618.70	1618.70	1618.70	1618.70	1618.70	1618.70	1618.70	...
3	9	1149.20	1149.20	1149.20	1149.20	1149.20	1149.20	1149.20	1149.20	1149.20	...
4	11	602.64	602.64	602.64	602.64	602.64	602.64	602.64	602.64	602.64	...
5	13	1331.66	1331.66	1331.66	1331.66	1331.66	1331.66	1331.66	1331.66	1331.66	...
6	14	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...
7	16	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...
8	18	517.64	517.64	517.64	517.64	517.64	517.64	517.64	517.64	517.64	...
9	23	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...

Reading Data with Pandas

Code

Output

```
df_numeric.iloc[ 1:10, 300:310]
```

>>>

	L1_S24_F1386	L1_S24_F1391	L1_S24_F1396	L1_S24_F1401	L1_S24_F1406	L1_S24_F1411	L1_S24_F1416	L1_S24_F1421	L1_S24_F1426	L1_S24_F1431
1	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
3	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
4	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
5	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
6	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
7	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
8	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
9	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

```
df_date.iloc[ 1:10, 300:310]
```

>>>

	L1_S24_D1151	L1_S24_D1153	L1_S24_D1155	L1_S24_D1158	L1_S24_D1163	L1_S24_D1168	L1_S24_D1171	L1_S24_D1173	L1_S24_D1175	L1_S24_D1178
1	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
3	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
4	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
5	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
6	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
7	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
8	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
9	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

Data Preprocessing

- Total number of Numeric features : 968
- Response = 1 for defective item
- Response = 0 for non-defective item
- Date data has 1157 columns
- More than 80% of date columns have missing values
- Most of the stations possess the same timestamp
- Evaluate Numeric feature data for Not a Number(NAN)
- Find the columns that have only NANs
- Find columns that have some NANs
- Impute Data into columns with NANs using mean value

Data Split

- Separate the Features and response as X and y

```
X = df_numeric[features].values
y = df_numeric["Response"].values
```

- Train and test split

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3)
```

Performance measures (tailored for this dataset)

Confusion matrix

		Predicted	
		Negatives (0)	Positives (1)
Actual	Negatives (0)	TN	FP
	Positives (1)	FN	TP

Terms

1. True Positives(TP) - Actual class was 1(True) and Predicted class is also 1 (True)
2. True Negatives(TN) - Actual class was 0(False) and Predicted class is also 0 (False)
3. False Positives(FP) - Actual class was 0(False) and Predicted class is 1(True)
4. False Negatives(FN) - Actual class was 1(True) and Predicted class is 0(False)

Performance measures

		Predicted	
		Negatives (0)	Positives (1)
Actual	Negatives (0)	TN	FP
	Positives (1)	FN	TP

- Metrics considered to decide the feature selection method for classification

  1. Accuracy $= \frac{TP+TN}{TP+FP+FN+TN}$ (No. of correct predictions/Total predictions)

  2. Precision $= \frac{TP}{TP+FP}$ (No. of correct positive predictions/Total positive predictions)

  3. Recall $= \frac{TP}{TP+FN}$ (No. of relevant positive predictions/Total actual positives)

  4. F1 Score $= 2 \times Precision \times Recall / (Precision + Recall)$

  5. Support - Number of samples of the true response that lie in each class

Feature Selection

- Using Ensemble methods to select the features that contribute to the Prediction

  1. Extra Trees Classifier

  2. Random Forest Classifier

  3. Gradient Boosting Classifier

Feature Selection

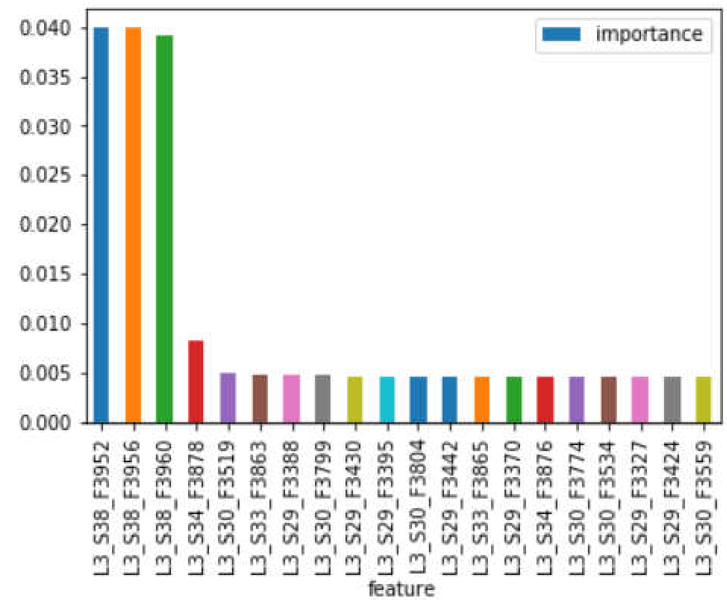
- Selection using Extra Trees Classifier

```
xt = ExtraTreesClassifier(n_estimators=10, verbose=2)

xt.fit(X_train, y_train)

Prediction = xt.predict(X_test)
```

- Classification Report

Accuracy : 99.425

Response	Precision	Recall	F1-Score	support
0	0.994	1.000	0.997	353069
1	0.688	0.005	0.011	2056
Avg/Total	0.992	0.994	0.991	355125



Top 20 Features based on Extra Trees results

Feature Selection

- Selection using Random Forest Classifier

  rfc = RandomForestClassifier(n_estimators=10, verbose=2)

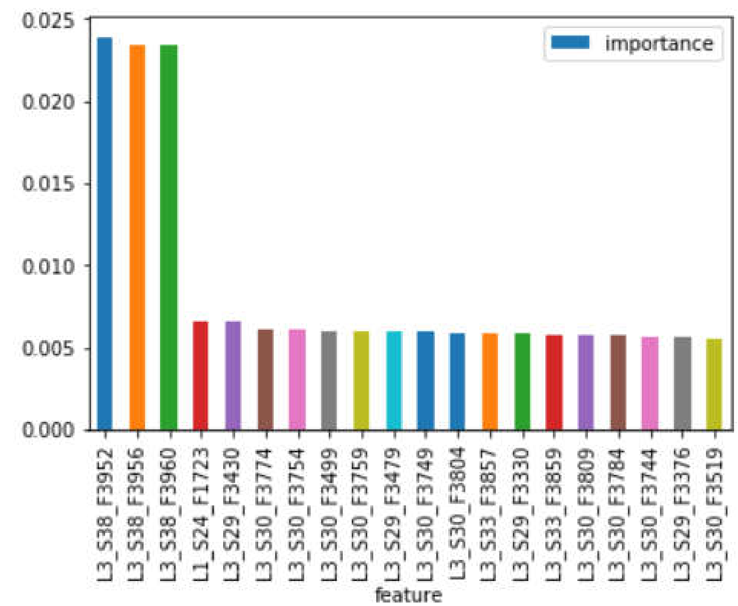
  rfc.fit(X_train, y_train)

  Prediction = rfc.predict(X_test)

- Classification Report

  Accuracy : 99.4249912002

Response	Precision	Recall	F1-Score	support
0	0.994	1.000	0.997	353069
1	0.590	0.022	0.043	2056
Avg/Total	0.992	0.994	0.992	355125



Top 20 Features based on Random Forest results

Feature Selection (contd..)

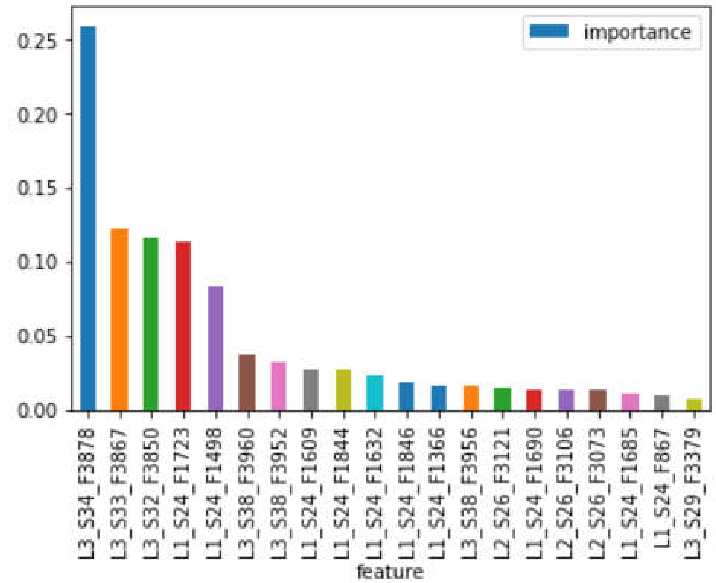
- Selection using Gradient Boosting Classifier

```
gbc = GradientBoostingClassifier(n_estimators=10, verbose=2)

gbc.fit(X_train, y_train)

Prediction = gbc.predict(X_test)
```

- Classification Report

Accuracy : 99.4340021119

Response	Precision	Recall	F1-Score	support
0	0.995	1.000	0.997	353069
1	0.634	0.053	0.098	2056
Avg/Total	0.992	0.994	0.992	355125



Top 20 Features based on Gradient Boost results

Model Training and Inference

- Merge Key features from the Random Forest and Gradient Boosting classifiers
  - filtered_feature_list = list(set(rf_selectfrommodel + gb_selectfrommodel))
- Create a new Data frame with the selected features (subset)
  - X_new = df_numeric[filtered_feature_list].values
- Split the new data frame to train and test
  - X_new_train, X_new_test = train_test_split(X_new, test_size=0.3)
- Models evaluated for training and Inference
  1. Random Forest
  2. Gradient Boost
  3. LinearSVC

Model Training and Inference – Random Forest

Train and test with Random Forest Classifier

rf_model = RandomForestClassifier(n_estimators=100, verbose=2)

rf_model.fit(X_new_train, y_train)

prediction = rf_model.predict(X_new_test)

Classification Report

Accuracy : 99.4354100669

Response	Precision	Recall	F1-Score	support
0	0.994	1.000	0.997	353069
1	0.892	0.028	0.055	2056
Avg/Total	0.994	0.994	0.992	355125

The precision at 89.2%, Random Forest is a reasonably good model with less false positives.

Model Training and Inference – Gradient Boost

Train and test with Gradient Boosting Classifier

```
gb_model = GradientBoostingClassifier(n_estimators=100, verbose=2)
gb_model.fit(X_new_train, y_train)
prediction = gb_model.predict(X_new_test)
```

Classification Report

Accuracy : 99.4280887012

Response	Precision	Recall	F1-Score	support
0	0.994	1.000	0.997	353069
1	0.577	0.046	0.085	2056
Avg/Total	0.992	0.994	0.992	355125

The precision at 57.7%, Gradient Boost has high false positives compared with Random Forest

Model Training and Inference – LinearSVC

Train and test with Linear Support Vector Machine

```
lsvm_model = LinearSVC(verbose=2)

lsvm_model.fit(X_new_train, y_train)

prediction = lsvm_model.predict(X_new_test)
```

Classification Report

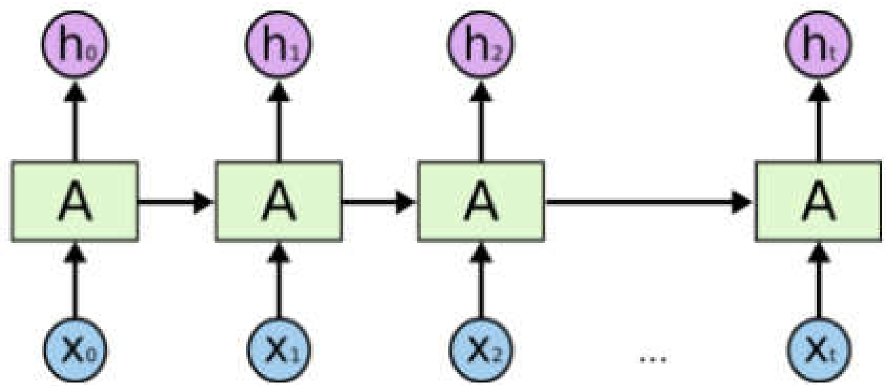
Accuracy : 99.4227384724

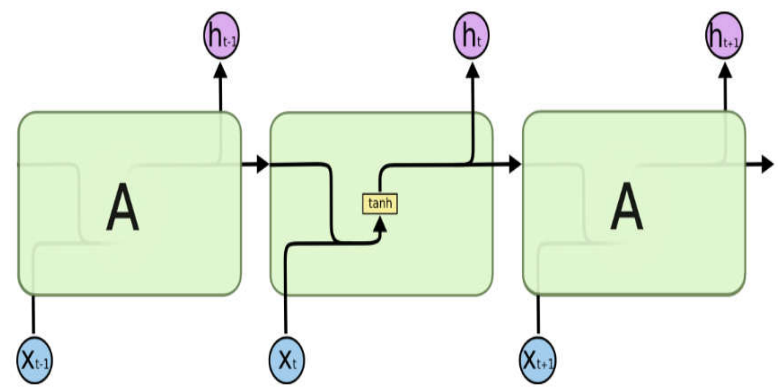
Response	Precision	Recall	F1-Score	support
0	0.994	1.000	0.997	353069
1	0.650	0.006	0.013	2056
Avg/Total	0.992	0.994	0.991	355125

The precision at 65%, LinearSVC has high false positives compared with Random Forest

Recurrent Neural Networks

Learning from persisted information - Understanding from previous state



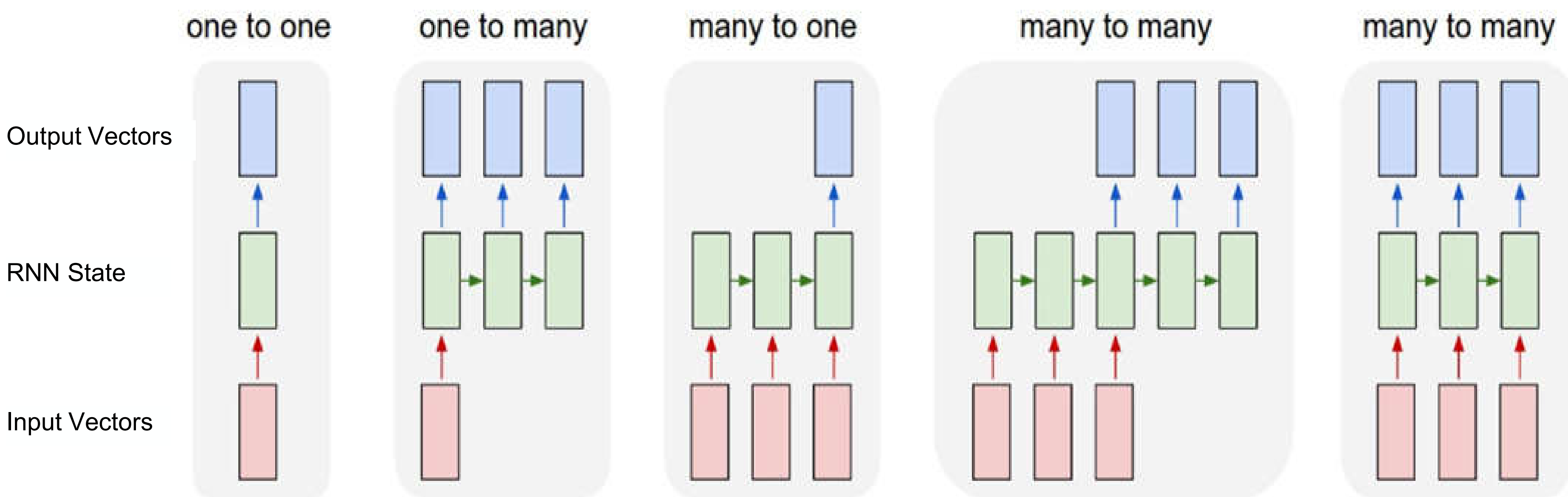


The module A gets input X_i and a looped information from previous modules in the network

The module A is a repeating throughout the network

Operate on Sequences of vectors: Sequences in the input, the output, or both

Recurrent Neural Networks



No RNN | Sequence output | Sequence Input | Sequence Input and output | Synced sequence input and output

Image Classification | Image Captioning | Sentiment Analysis | Machine Translation | Video Classification

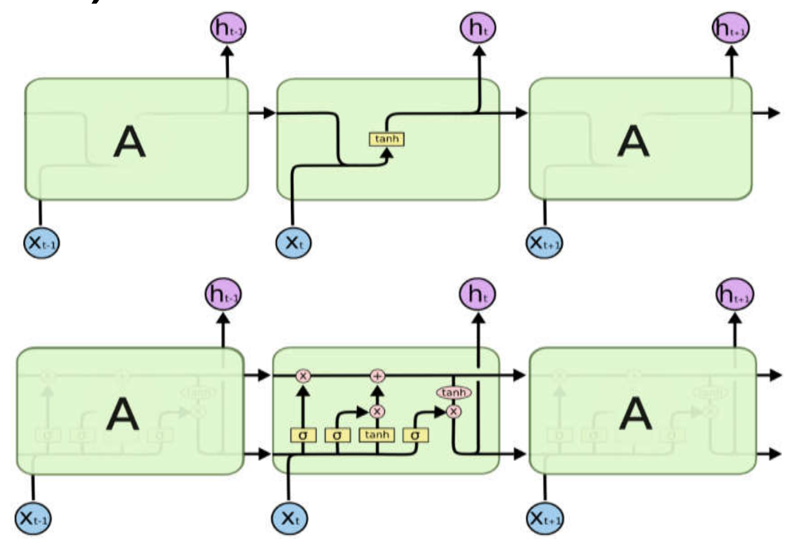
Long Short Term Memory(LSTM) Networks

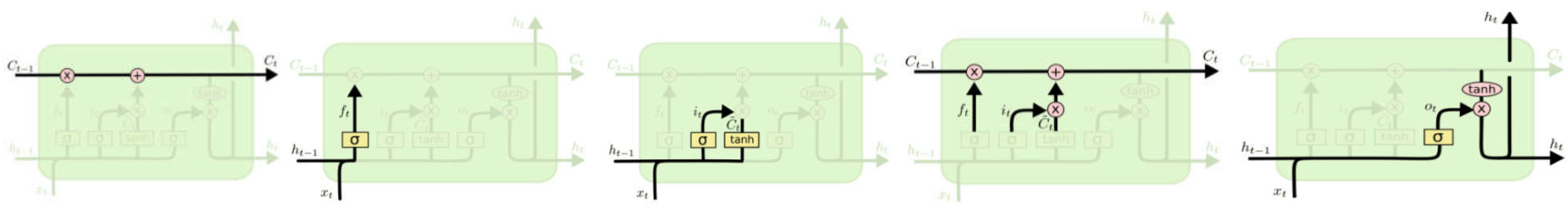
RNNs fail to handle long term dependencies

In RNN usually the repeating module A has a simple structure consisting of a single layer

LSTM's designed to remember information for a long periods

In LSTMs the repeating module consists of 4 interacting layers





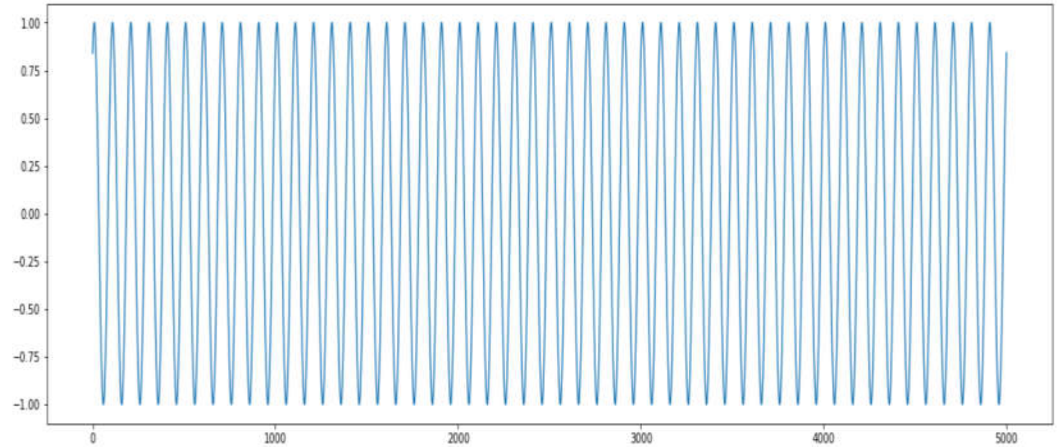
LSTM at Work – Sine wave example

Read the Sine Wave input

```
series = pd.read_csv('sine-wave.csv', header=None)
series.head(4)
```

	0
0	0.841471
1	0.873736
2	0.902554
3	0.927809

```
pyplot.plot(series.values)
pyplot.show()
```



First n data points used as Input (X) to predict y1 the n+1 data point

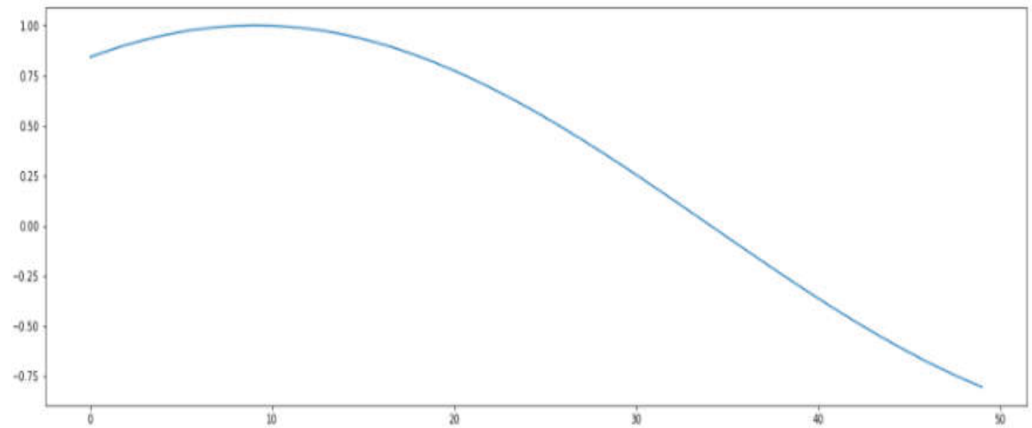
Use the window between 1 to n+1 data points as input to predict y2 the n+2 data point

Use a 2 layered LSTM architecture to make the prediction

LSTM at Work – Sine wave example

First 50 point wave plot

```
pyplot.plot(series.values[:50])
pyplot.show()
```



Fix the moving window size to 50 -> Keep shifting the entire column and concatenate to the series

LSTM at Work – Sine wave example

Data Split

Split the series data set into train and test

Train data at 80% and Test data at 20%

Take first 50 data points as X and 51st point at y

Create X and y train and test sets

```
nrow = round(0.8*series.shape[0])
train = series.iloc[:nrow, :]
test = series.iloc[nrow:,:]
train_X = train.iloc[:,:-1]
train_y = train.iloc[:,-1]
test_X = test.iloc[:,:-1]
test_y = test.iloc[:,-1]
```

LSTM Model with sample code

```
model = Sequential()
model.add(LSTM(input_shape = (50,1), output_dim= 50, return_sequences = True))
model.add(Dropout(0.5))
model.add(LSTM(256))
```

LSTM at Work – Sine wave example

Train and predict

Compile the model

```
model.compile(loss="mse", optimizer="adam")
```

Train and predict

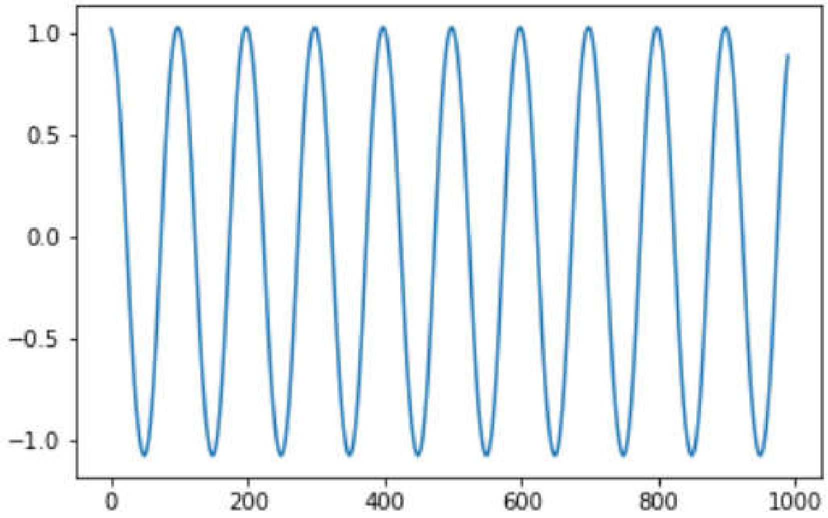
```
model.fit(train_X,train_y,batch_size=512,nb_epoch=3,validation_split=0.1)
preds = model.predict(test_X)
actuals = test_y
mean_squared_error(actuals,preds)
```

```
Out[38]: 0.003095152635107611
```

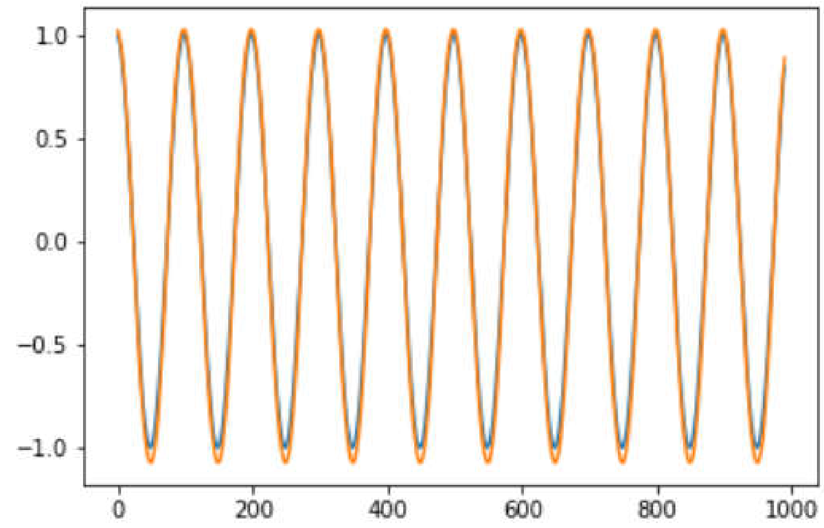
LSTM at Work – Sine wave example

Plot actual vs predicted

```
pyplot.plot(preds)
pyplot.show()
```




```
pyplot.plot(actuals)
pyplot.plot(preds)
pyplot.show()
```



Software