# Explorations in Data Science

Cisco Systems

## Abstract

Whereas Cisco today have hardware based platforms such as AI/ML ready HyperFlex with TensorFlow and NVIDIA GPU support, implementing ML based models across customer networks will require significant understanding of the customer's requirements, data characteristics and specific use case – this deliverable, in turn will essentially involve making each customer deployment a unique, new customized ML driven environment, be that Health Services such as UK's NHS or Financial Research communities such as Bloomberg. Keeping the importance of both health and financial sectors in mind, this study focuses on exploring the following two use cases and datasets: (1) A ML driven study and prediction model of cryptocurrency (bitcoin) and (2) An exploratory study of the Wisconsin Breast cancer dataset, made public.

Santanu Ganguly (santagan)
santagan@cisco.com

# Explorations in Data Science

Santanu Ganguly (santagan@cisco.com)

01st Draft

Cisco Systems, UK

| Version Control | | |
|---|---|---|
| Date | Version | Comments |
| 25 June 2019 | 01st Draft | S. Ganguly |
| | | |
| | | |
| | | |

# Contents

# 1 INTRODUCTION

The advent of Big Data saw most of the enterprise and service provider space join the Data Analytics and eventually Data Science revolution. Cisco Systems, in their endeavour to understand and lead the data driven transformation of customers' ecosystems, brought in due emphasis on exploration methodology of data inside of the company [1]. Accessing data sets is an ongoing challenge across the technology industry and Cisco. As individuals and teams seek to begin or continue their journey to increased data-driven decision making, access has been at best, challenging and at worst, show stopping.

## 1.1 BUSINESS MOTIVATION

Cisco's UK based Public Services pre-sales technical team observes, almost on a daily basis, data handling related challenges across various portfolios of customers spanning industries as diverse as Aerospace, Health, Financial and Academic Research organisations. Whether the data is health science related or involves challenges of understanding the prediction of financial markets or running machine algorithms such as CNN (Convolutional Neural networks) on astrophysical data to detect structures such as Quasars and Gravitational Lensing from massive data sizes, Cisco has customers approaching with queries about the most efficient solutions in their specific environment. Whereas Cisco today have hardware based platforms such as AI/ML ready HyperFlex with TensorFlow and NVIDIA GPU support, implementing ML based models across customer networks will require significant understanding of the customer's requirements, data characteristics and specific use case – this deliverable, in turn will essentially involve making each customer deployment a unique, new customized ML driven environment, be that Health Services such as UK's NHS or Financial Research communities such as Bloomberg.

One of the most important aspects of Cisco's current business model is intent driven networking. In order to have networks which are capable of self-regulating and self-healing, data analytics and ability to treat that same data in an efficient manner in a customized environment is of critical importance. Reduction of noise on that same data via selecting it correctly and cleaning it is of primary focus, because if we have noise on the data that we are training in a machine-learning environment, then we shall be training the noise too, which, in future will impact the prediction made by the same model.

## 1.2 USE CASES & DATASETS

Keeping the importance of both health and financial sectors in mind, this study focuses on exploring the following two use cases and datasets:

1) A ML driven study and prediction model of cryptocurrency (bitcoin).
2) An exploratory study of the Wisconsin Breast cancer dataset, made public.

## 1.3 STRUCTURING THE PROBLEM

The most important factor to consider when developing a datascience system is acquiring data that represents a given problem correctly, followed by the understanding of the datasets inherent biases and limitations. For this reason, it is often useful to reflect upon a decision tree at the start of a deep learning project, as depicted in Fig. 1 below

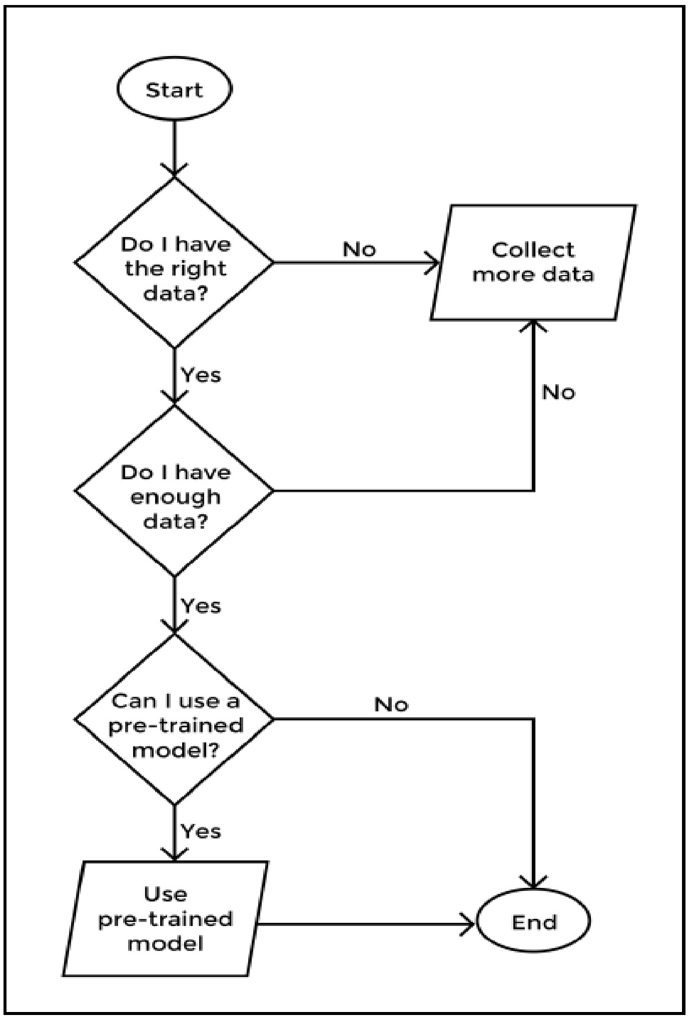


**Fig. 1**. Model decision tree beginning of a deep learning project

Following are some of the important aspects to consider when starting a Datascience project:

- **Selecting correct data**: This is the hardest challenge when training a deep learning model. Firstly, the problem needs to be defined with mathematical rules using precise definitions. The problem needs to be organized in either categories (classification problems) or a continuous scale (regression problems).
- **Sufficient data availability for the given problem**: Typically, deep learning algorithms have shown to perform much better in large datasets than in smaller ones. Amount of data necessary to train a high-performance algorithm depends on the kind of problem that is being addressed; however, the aim should be to collect as much data as possible.
- **Usage of a pre-trained model**: When working on a problem that is a subset of a more general application — but within the same domain, using a pretrained model can give a head start on tackling the specific patterns of the problem, instead of the more general characteristics of the domain at large. A good place to start is the official TensorFlow repository (https://github. com/tensorflow/models) for generic data. For more specific data, for example, CNN (Convolutional Neural Networks) training of Astrophysical Gravitational Lenses, a good place to start would be the Ensai Project: https://github.com/yasharhezaveh/Ensai . The later instance (Ensai) is an example of a case where enough data is very challenging to come by but it is imperative for us to try, identify

and characterise Gravitational Lenses. In this case, a series of techniques were used to effectively create more data synthetically from an input of an unlensed image. This process is known as **data augmentation** and has successful application in image recognition.

## 1.4 NN: Model Evaluation, Optimization & Hyperparameter Tuning

This section focusses on the evaluation of the neural network (NN) model.

### 1.4.1 Model Evaluation & Problem Categories

While working with NNs, the network's performance is measured and then its hyperparameters [4] are modified to improve its performance. "Hyperparameters" are parameters in an NN which cannot be learned like regular parameters such as weights and biases but have to be adjusted by hand. An example of a hyperparameter is the hidden layer size or learning rate of a weighted dataset.

Problems and associated solutions by NNs are categorized into two:

- **Classification**: In classification problems [5] categories can be different or similar; they can also be about a binary problem. However, they must be clearly assigned to each data element. An example of a classification problem would be to assign the label *car* or *not car* to an image using a Convolutional Neural Network (CNN).
- **Regression**: Regression problems [5] are characterized by a continuous variable, that is, a scalar. These problems are measured in terms of ranges, and their evaluations regard how close to the real values the network is. An example is a time-series classification problem in which a Recurrent Neural Network (RNN) is used to predict the future temperature values. The Bitcoin price-prediction problem is an example of a regression problem.

### 1.4.2 Loss Functions, Error Rates & Accuracy

Neural networks utilize functions that measure how the networks perform when compared to a validation set—that is, a part of the data separated to be used as part of the training process. These functions are known as **loss functions**.

**Loss functions** evaluate how wrong a neural network's predictions are. They propagate those errors back with adjustments made to the network, modifying how individual neurons are activated. Loss functions are key components of neural networks and choice of the loss function can have a significant impact on how the network performs.

In an NN, **error propagation** happens via a process called back propagation. Back propagation is a technique for propagating the errors returned by the loss function back to each neuron in a neural network. Propagated errors affect how neurons activate, and ultimately, how they influence the output of that network. Several NN packages, including Keras, use this technique by default.

Loss functions for used or regression and classification problems tend to be different. For classification problems, accuracy functions (i.e., the proportion of times the predictions were correct) is usually engaged. While for regression problems, error rates (i.e., how close the predicted values were to the observed ones) is used.

The following table 1 provides a summary of common loss functions to utilize, alongside their common applications:

**Table 1: Summary of common loss functions**

| Problem Type | Loss Function | Problem | Example Use Cases |
|---|---|---|---|
| Regression | Mean Squared Error (MSE) | Predicting of a continuous function, i.e., predicting value within a range of values. | Predicting the temperature in the future using temperature measurements from the past. |
| Regression | Root Mean Squared Error (RMSE) | Deals with negative values. Predicting of a continuous function, i.e., predicting value within a range of values. Typically provides more interpretable results | Predicting the temperature in the future using temperature measurements from the past. |
| Regression | Mean Absolute Percentage error | Predicting of continuous functions. Better performance when working with de-normalized ranges. | Predicting the sales for a product using the product properties (for example, price, type, target audience, market conditions). |
| Classification | Binary Cross Entropy | Classification between two categories or between two values (that is, true or false). | Predicting if the visitor of a website is male or female based on their browser activity. |
| Classification | Categorical Cross Entropy | Classification between many categories from a known set of categories. | Predicting nationality of a speaker based on their accent when speaking a sentence in English. |

For *regression* problems, the MSE function is the most common choice. While for classification problems, Binary Cross-entropy (for binary category problems) and Categorical Cross-entropy (for multi-category problems) are common choices. Generally related projects  start with these loss functions, then experiment with other functions as the neural network evolves, aiming to gain performance.

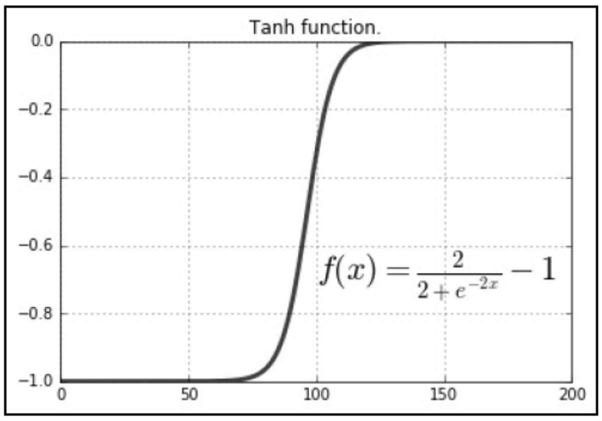
### 1.4.3 Activation Functions

Activation functions [2]  determine the value that each neuron will pass to the next element of the network, using both the input from the previous layer and the results from the loss function—or if a neuron should pass any values at all.

TensorFlow and Keras provide many activation functions—and new ones are occasionally added. Following three are important considerations in context of the current project:

- **Linear:** Linear functions only activate a neuron based on a constant value defined by:
$$f(x) = c * (0, x)$$
when $c = 1$ neurons will pass the values as is. The issue with using linear functions is that due to the fact that neurons are activated linearly, chained layers tend to function as a single large layer. In other words, the model loses the ability to construct networks with many layers, in which the output of one influences the other.

- **Hyperbolic Tangent (tanh)**: Tanh is a non-linear function represented by the following:
$$f(x) = \frac{2}{2 + e^{-2x}} - 1$$

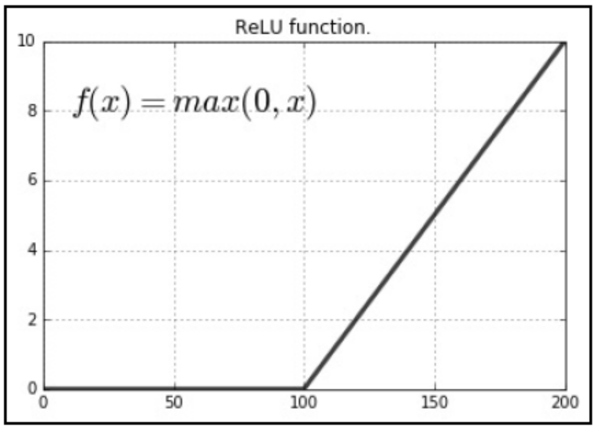


***tanh function***

The effect *tanh* has on nodes is evaluated continuously. Also, because of its non-linearity, one can use this function to change how one layer influences the next layer in the chain. When using non-linear functions, layers activate neurons in different ways, making it easier to learn different representations from data. However, they have a sigmoid-like pattern which penalizes extreme node values repeatedly, causing a problem called vanishing gradients. Vanishing gradients have negative effects on the ability of a network to learn.

- **Rectifid Linear Unit (ReLU):** ReLUs have non-linear properties. They are defined as follows:

$$f(x) = max(0, x)$$



***ReLU Function***

ReLU functions are often recommended as great starting points before trying other functions. ReLUs tend to penalize negative values. So, if the input data (for instance, normalized between -1 and 1) contains negative values, those will now be penalized by
ReLUs.

# 2 Project 1: Bitcoin Dataset

This section of the project will explore publicly available bitcoin dataset retrieved from CoinMarketCap ( https://coinmarketcap.com/ ), a popular website which tracks different cryptocurrency statistics. The dataset `bitcoin_prices.csv` contains Bitcoin prices from early 2013 till November 2017. This data was made publicly available by CoinMarketCap only for the above duration of time period.

## 2.1 Data Exploration: Software Used

Table 2. below shows software components were used for this Project:

**Table 2. Software used**

| Software | Description | Version |
|---|---|---|
| Python | Programming language | 3.7.3 |
| TensorFlow | Graph computation Python package open-sourced by Google in 2017, typically used for developing deep learning systems | 1.14 |
| Keras | Python package which provides a high-level interface to TensorFlow. | 2.0.8-tf |
| TensorBoard | Browser-based software for visualizing neural network statistics. | 1.13.1 |
| Jupyter Notebook | Browser-based software for working interactively with Python sessions. | 5.7.8 |
| Pandas | Python package for analysing and manipulating data. | 0.21.0 |
| Numpy | Python package for high-performance numerical computations. | 1.13.3 |

### 2.1.1 Data Acquisition & Data Characteristics

The data is loaded into Jupyter notebook via Pandas dataframe in order to ease the cleaning process. The dataset contains 8 variables (i.e. columns) as follows with the values in US dollars:

- `` `date` ``: date of the observation.
- `` `iso_week` ``: week number of a given year.
- `` `open` ``: open value of a single Bitcoin coin.
- `` `high` ``: highest value achieved during a given day period.
- `` `low` ``: lowest value achieved during a given day period.
- `` `close` ``: value at the close of the transaction day.
- `` `volume` ``: what is the total volume of Bitcoin that was exchanged during that day.
- `` `market_capitalization` ``: as described in CoinMarketCap's FAQ page, this is calculated by Market Cap = Price X Circulating Supply.

Two of the variables of the `data, date` and `iso_week`, can be used as indices and the rest of the six can be used to understand how the value of Bitcoin has changed over time.

### 2.1.2 Data Cleaning & Exploration

As a next step, the dataset timeseries is explored in order to understand its patterns. The dataset `bitcoin_prices.csv` contains Bitcoin prices from early 2013 till November 2017. This data was made publicly available by CoinMarketCap *only for the above duration of time period*.

> **Please Note**: CoinMarketCap (https://coinmarketcap.com/) is a great resource for crypto prices and data. However, downloading of historical cryto data in order to do some data analysis is not very straight forward from a legal perspective. Although in the past they had a

public API, this never had the facility for downloading any historical data. There is also a notice that this API was taken offline in Dec 2018 and replaced by a Professional API. The Professional API allows for historical data to be downloaded but at the very steep price of $699/month for access to 12 months of historical data and uses need to “inquire for pricing” for access to up to 5 years of historical data.

A script was constructed to download the data only and testing proved it to be working correctly. However, fortunately, the author of this report decided to scan the terms of historical data use governing the site first. It is clearly stated there, in no uncertain terms, that users are not allowed to leaglly scrape or copy any data from the site. Following is an extract from the Terms of Use :

```
Prohibited Activities

You agree that you will not:

Use or introduce to the Service any data mining, crawling,
“scraping”, robot or similar automated or data gathering or
extraction method, or manually access, acquire, monitor or copy any
portion of the Service, or download or store Content (unless
expressly authorized by CMC). Certain data and other information
within the Service is available by subscription, or for a fee, at
https://pro.coinmarketcap.com;
```

Strictly hypothetically speaking, *if the above legality would not be a constraint*, then following would be a way to obtain the data in Python:

1) As a first step, BeautifulSoup and a few other packages would be imported:

```
from bs4 import BeautifulSoup
import requests
import pandas as pd
```

2) Then, the relevant page will be fetched and the table that contains all the data would be extracted:

```
url = "https://coinmarketcap.com/currencies/ripple/historical-
data/?start=20130428&end=20180802"
content = requests.get(url).content
soup = BeautifulSoup(content,'html.parser')
table = soup.find('table', {'class': 'table'})
```

3) Step three would be a nested list comprehension to go through the table rows and cells building up a list of lists with the data:

```
data = [[td.text.strip() for td in tr.findChildren('td')]

            for tr in table.findChildren('tr')]
```

4) **Cleaning**: Finally, the list of lists is converted to a DataFrame and some **cleaning** up done such as converting text dates to real dates, removing empty rows, naming the columns, setting an index, and sorting.

```
df = pd.DataFrame(data)
df.drop(df.index[0], inplace=True) # first row is empty
df[0] =  pd.to_datetime(df[0]) # date
for i in range(1,7):
    df[i] =
pd.to_numeric(df[i].str.replace(",","").str.replace("-","")) #
some vol is missing and has - df.columns =
['Date','Open','High','Low','Close','Volume','Market Cap']
df.set_index('Date',inplace=True)
df.sort_index(inplace=True)
```

At a first instance two variables are explored: `close` price and `volume`: `volume` contains data starting in November 2013, while `close` prices start earlier in April of the same year. However, both show similar spiking patterns starting at the beginning of 2017 as shown in Fig. 2a below

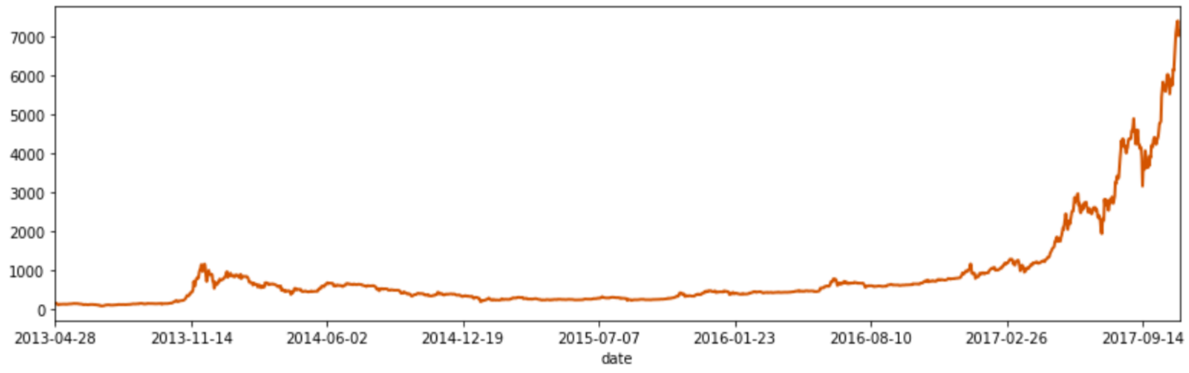


**Fig. 2a**. Time series plot of close prices of Bitcoin from the `close` variable showing variation with date

The early spike in Fig. 2a in 2013-2014 shows the initial price rush and then the recent spike in 2017 shows how the prices have continued to skyrocket, despite trenches from time to time. The overall value prospect of Bitcoin remains strongly positive.

Next, a similar plot was generated for the same date range for the `volume` variable as shown in Fig 2b.

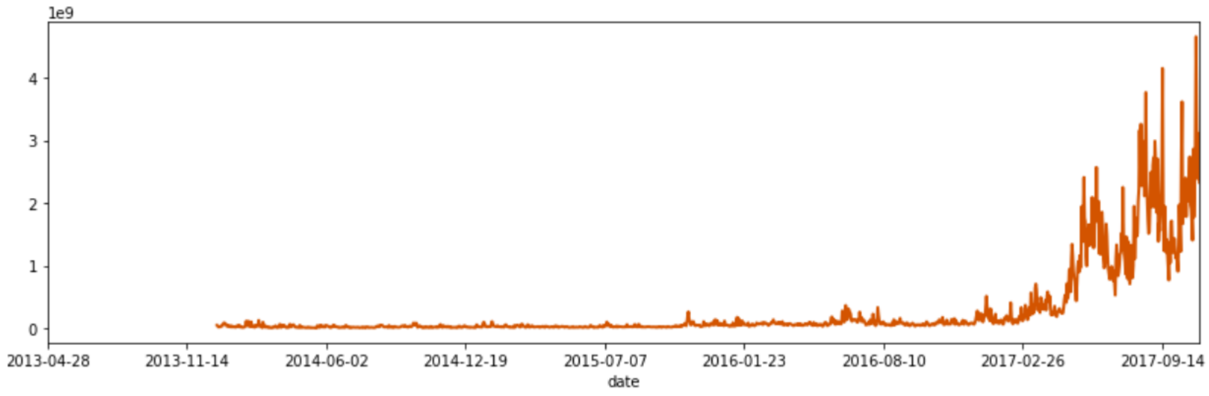


**Fig. 2b**. Time series plot of `volume` variable of Bitcoin showing variation with date

The trend in Fig. 2b shows that starting 2016-2107 a trend starts in which a significantly larger amount (volume) of Bitcoin is being traded in the market. The total daily volume varies much more than the daily price closes.

The above two plots reflect the critical phenomenon that both the prices and volume of Bitcoin have been continuously growing since 2016-2017.

For a closer look, data of year of 2017 only was explored as this is the year where the price of bitcoin had risen significantly. Plots for both close price (Fig. 2c) and volume (Fig. 2d) were generated as below:

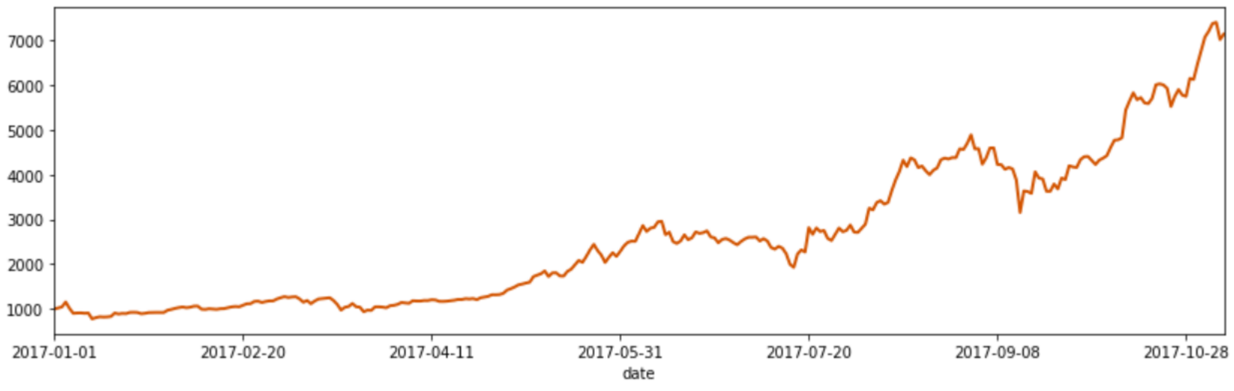


**Fig. 2c**. Time series plot of `close` variable of Bitcoin showing variation for year 2017 only

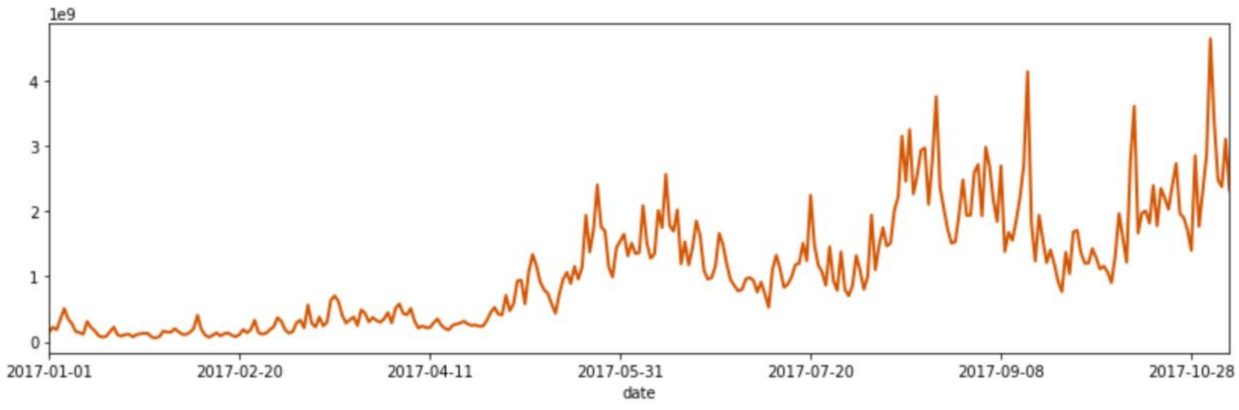
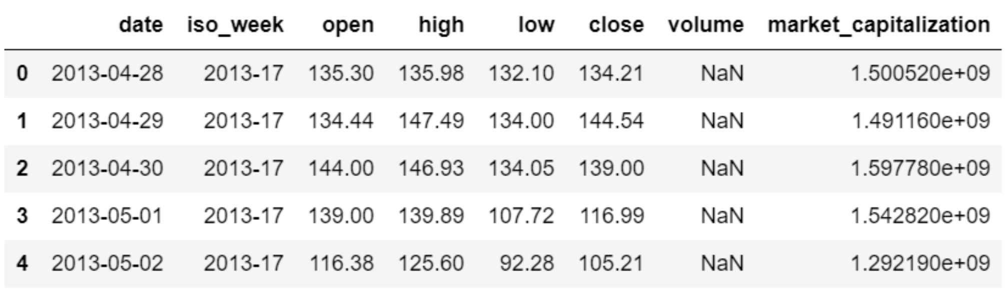


**Fig. 2d**. Time series plot of `volume` variable of Bitcoin showing variation for year 2017 only

### 2.1.3 Preparation of Dataset for Modelling

*Neural networks* typically work with either matrices or tensors. The Bitcoin data needs to fit that structure before it can be made fit to be usable by either keras or tensorflow. Also, it is common practice to normalize data before using it to train a neural network. The custom `normalization.py` function will be used which will evaluate each observation into a range between 0 and 1 in relation to the first observation of each week.

The Python `head()` method was used to obtain a quick glimpse of the dataset (Fig. 3):



| | date | iso_week | open | high | low | close | volume | market_capitalization |
|---|---|---|---|---|---|---|---|---|
| **0** | 2013-04-28 | 2013-17 | 135.30 | 135.98 | 132.10 | 134.21 | NaN | 1.500520e+09 |
| **1** | 2013-04-29 | 2013-17 | 134.44 | 147.49 | 134.00 | 144.54 | NaN | 1.491160e+09 |
| **2** | 2013-04-30 | 2013-17 | 144.00 | 146.93 | 134.05 | 139.00 | NaN | 1.597780e+09 |
| **3** | 2013-05-01 | 2013-17 | 139.00 | 139.89 | 107.72 | 116.99 | NaN | 1.542820e+09 |
| **4** | 2013-05-02 | 2013-17 | 116.38 | 125.60 | 92.28 | 105.21 | NaN | 1.292190e+09 |

**Fig. 3**. Python `head()` method gives a glimpse of the Bitcoin data

Data cleanup and filtering: Data was removed from older time periods keeping only the data from 2016 until the latest observation of 2017. Whereas older observations may be useful to understand current prices, Bitcoin's increased popularity in recent years makes inclusion of older data a more laborious and computationally expensive treatment which can be considered for future exploration.

The following Pandas API was used for filtering the data for years 2016 and 2017:

```
bitcoin_recent = bitcoin[bitcoin['date'] >= '2016-01-01']
```

Only the `close` and `volume` variables were retained for this study:

```
bitcoin_recent = bitcoin_recent[['date', 'iso_week', 'close', 'volume']]
```

The variable `bitcoin_recent` now has a copy of the original bitcoin dataset, but filtered to the observations that are newer or equal to January 1, 2016.

As a final step, the data was normalized using the point-relative normalization technique. Only two variables were normalized (`close` and `volume`), because those are the variables that we are working to predict. Research have proposed that normalization is an essential technique for training RNNs and LSTMs, mainly because it decreases the network's training time and increases the network's overall performance.

Normalization of the data for the `close` variable:

```
bitcoin_recent['close_point_relative_normalization'] =
bitcoin_recent.groupby('iso_week')['close'].apply( lambda x:
normalizations.point_relative_normalization(x))
```

The variable `'close_point_relative_normalization'` now contained the normalized data for the variable `close.`

The same normalization was applied on the `volume` variable:

```
bitcoin_recent['volume_point_relative_normalization'] =
bitcoin_recent.groupby('iso_week')['close'].apply(lambda x:
normalizations.point_relative_normalization(x))
```

After the normalization procedure, the variables ``close`` and ``volume`` were now relative to the first observation of every week.

The outputs of the above normalizations were stored in the following new variables: ``close_point_relative_normalization`` and ``volume_point_relative_normalization`` respectively to train the LSTM (Long Short-Term Memory) model.

The normalized `close` variable exhibited an interesting variance pattern on a weekly basis a shown in Fig. 4a below.

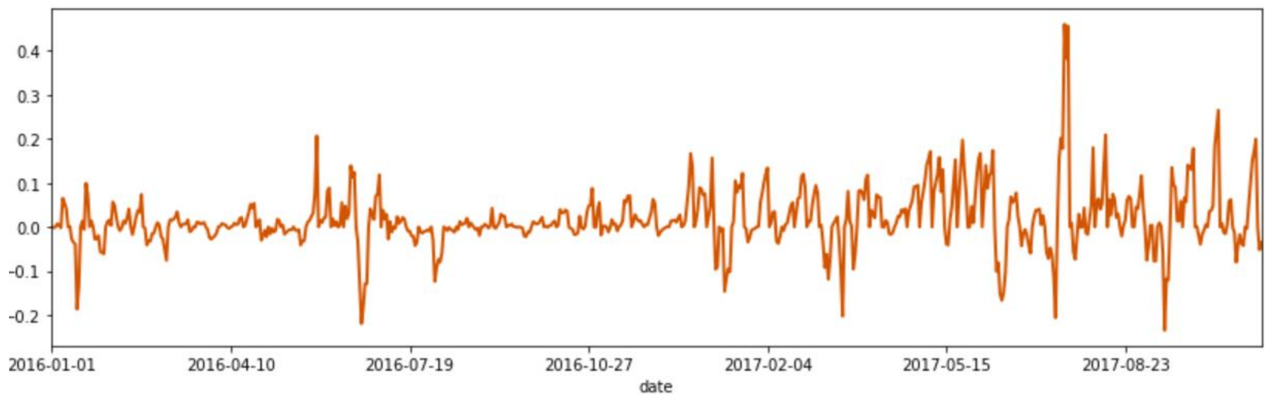


**Fig. 4a**. Normalized `close` variable showing the variance of the Bitcoin data

The normalized `volume` variable exhibited an interesting variance pattern on a weekly basis a shown in Fig. 4b below.

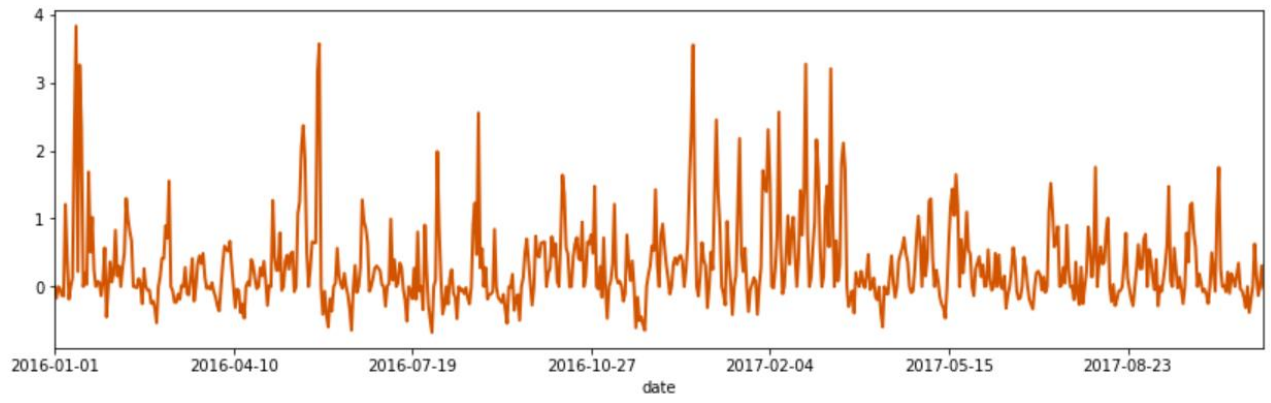
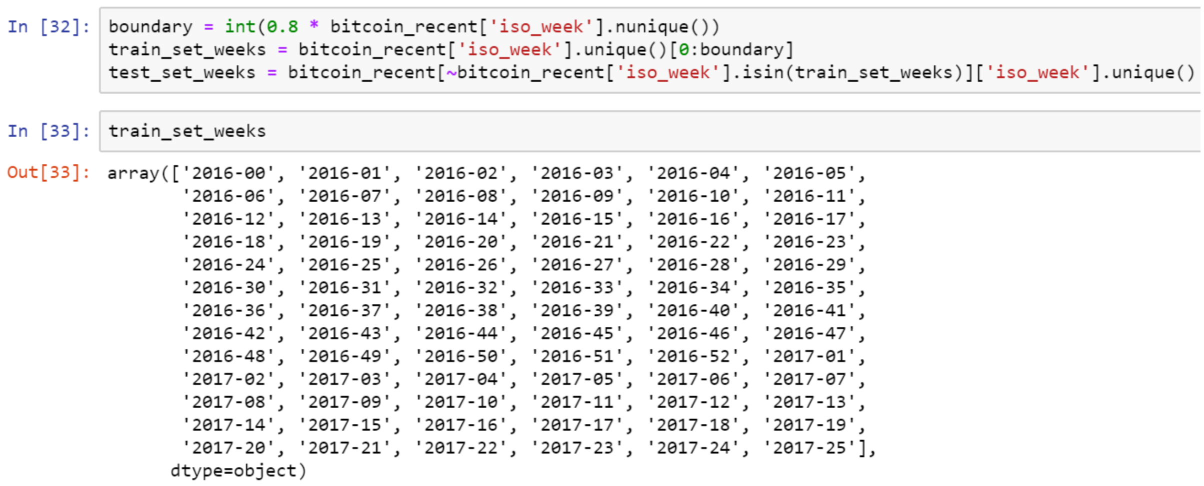


**Fig. 4b**. Normalized `volume` variable showing the variance of the Bitcoin data

### 2.1.4 Training & Test Sets

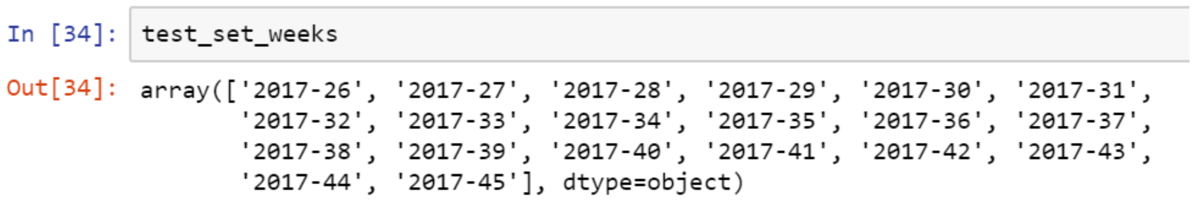
The outputs of the above normalizations were stored in the following new variables: `close_point_relative_normalization` and `volume_point_relative_normalization` respectively to train the LSTM (Long Short-Term Memory) model.

The dataset was divided into a training and a test set. In this case, 80% of the dataset was used to train the LSTM model and 20% to test its performance. Given that the data is continuous, last 20% of available weeks was used as a test set and the first 80% as a training set a shown in Fig. 5a & Fig. 5b of the Jupyter notebook screenshot below.



```
In [32]: boundary = int(0.8 * bitcoin_recent['iso_week'].nunique())
         train_set_weeks = bitcoin_recent['iso_week'].unique()[0:boundary]
         test_set_weeks = bitcoin_recent[~bitcoin_recent['iso_week'].isin(train_set_weeks)]['iso_week'].unique()

In [33]: train_set_weeks

Out[33]: array(['2016-00', '2016-01', '2016-02', '2016-03', '2016-04', '2016-05',
                '2016-06', '2016-07', '2016-08', '2016-09', '2016-10', '2016-11',
                '2016-12', '2016-13', '2016-14', '2016-15', '2016-16', '2016-17',
                '2016-18', '2016-19', '2016-20', '2016-21', '2016-22', '2016-23',
                '2016-24', '2016-25', '2016-26', '2016-27', '2016-28', '2016-29',
                '2016-30', '2016-31', '2016-32', '2016-33', '2016-34', '2016-35',
                '2016-36', '2016-37', '2016-38', '2016-39', '2016-40', '2016-41',
                '2016-42', '2016-43', '2016-44', '2016-45', '2016-46', '2016-47',
                '2016-48', '2016-49', '2016-50', '2016-51', '2016-52', '2017-01',
                '2017-02', '2017-03', '2017-04', '2017-05', '2017-06', '2017-07',
                '2017-08', '2017-09', '2017-10', '2017-11', '2017-12', '2017-13',
                '2017-14', '2017-15', '2017-16', '2017-17', '2017-18', '2017-19',
                '2017-20', '2017-21', '2017-22', '2017-23', '2017-24', '2017-25'],
               dtype=object)
```

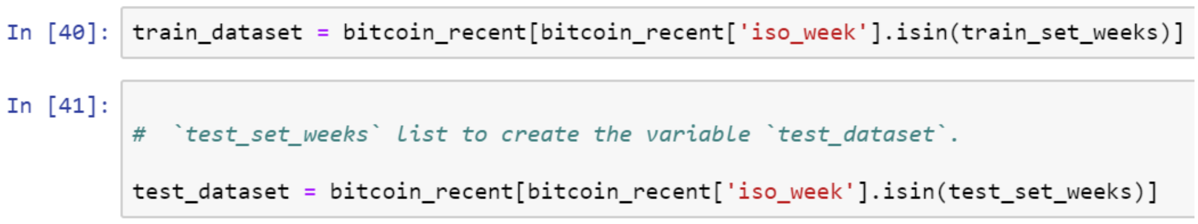
**Fig. 5a**. Weekly training data sets



```
In [34]: test_set_weeks

Out[34]: array(['2017-26', '2017-27', '2017-28', '2017-29', '2017-30', '2017-31',
                '2017-32', '2017-33', '2017-34', '2017-35', '2017-36', '2017-37',
                '2017-38', '2017-39', '2017-40', '2017-41', '2017-42', '2017-43',
                '2017-44', '2017-45'], dtype=object)
```

**Fig. 5b**. Weekly test data sets

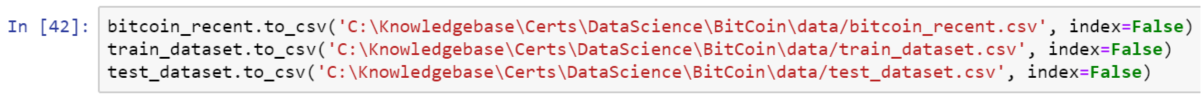
Separate datasets for training and test were created as shown in Fig 6:



```
In [40]: train_dataset = bitcoin_recent[bitcoin_recent['iso_week'].isin(train_set_weeks)]

In [41]:
         #  `test_set_weeks` list to create the variable `test_dataset`.

         test_dataset = bitcoin_recent[bitcoin_recent['iso_week'].isin(test_set_weeks)]
```

**Fig. 6**. Weekly training & test data sets

As a last step of this section, the outputs were stored on the hard-disk to make sure it is easier for ease use with this data as input to the neural network.



```
In [42]: bitcoin_recent.to_csv('C:\Knowledgebase\Certs\DataScience\BitCoin\data/bitcoin_recent.csv', index=False)
         train_dataset.to_csv('C:\Knowledgebase\Certs\DataScience\BitCoin\data/train_dataset.csv', index=False)
         test_dataset.to_csv('C:\Knowledgebase\Certs\DataScience\BitCoin\data/test_dataset.csv', index=False)
```

### 2.1.5 Conclusion

In this section, the Bitcoin dataset was explored. During the year of 2017 the prices of Bitcoin skyrocketed. This phenomenon takes a long time to take place—and may have been influenced by a number of external factors that this data alone doesn't explain (for instance, the emergence of other cryptocurrencies). The point-relative normalization technique was used to process the Bitcoin dataset into weekly chunks, which in turn will be used to train an LSTM network, a variant of Recurrent Neural Network (RNN), to learn the weekly patterns of Bitcoin price changes so that it can predict a full week in the future. Bitcoin statistics show significant fluctuations on a weekly basis despite an overall longer term upward trend. In the next section, an effort will be made to build a predictive model.

## 2.2 PREDICTIVE MODEL WITH TENSORFLOW AND KERAS

**Long-short term memory** (LSTM) networks are RNN variants created to address the vanishing gradient problem. The vanishing gradient problem is caused by memory components that are too distant from the current step and would receive lower weights due to their distance. LSTMs are a variant of RNNs that contain a memory component—known as *forget gate*. That component can be used to evaluate how both recent and old elements affect the weights and biases, depending on where the observation is placed in a sequence [2].

LSTM networks have input, hidden, and output layers. Each hidden layer has an activation function which evaluates that layer's associated weights and biases. The network moves data sequentially from one layer to another and evaluates the results by the output at every iteration, that is, per epoch.

LSTM networks are of interest because those networks perform well with sequential data—and time-series is a form of sequential data. Keras' `keras.models.Sequential()` component represents a whole sequential neural network of Python class which can be instantiated on its own, then have other components added to it subsequently.

Using Keras, the complete LSTM network would be implemented as follows:

```
from keras.models import Sequential
from keras.layers.recurrent import LSTM
from keras.layers.core import Dense, Activation
model = Sequential()
model.add(LSTM(
units=number_of_periods,
input_shape=(period_length, number_of_periods)
return_sequences=False), stateful=True)
model.add(Dense(units=period_length))
model.add(Activation("linear"))
model.compile(loss="mse", optimizer="rmsprop")
```

Keras abstraction allows for one to focus on the key elements that make a deep learning system more performant: what the right sequence of components is, how many layers and nodes to include, and which activation function to use. All of these choices are determined by either the order in which components are added to the instantiated `keras.models. Sequential()` class or by parameters passed to each component instantiation (that is, `Activation("linear")` ). The final `model.compile()` step builds the neural network using TensorFlow components. After the network is built, the network is trained using the `model.fit()` method. This yields a trained model that can be used to make predictions:

```
model.fit(
X_train, Y_train,
batch_size=32, epochs=epochs)
```

The variables `X_train` and `Y_train` are, respectively, a set used for training and a smaller set used for evaluating the loss function (that is, testing how well the network predicts data). Finally, we can make predictions using the `model.predict()` method:

```
model.predict(x=X_train)
```

Keras' paradigm allows for working with neural networks (NN) where different architectures can be dealt with in very different ways. Keras simplifies the interface for working with different architectures by using three components: network architecture, fit, and predict as shown in Fig. 7.

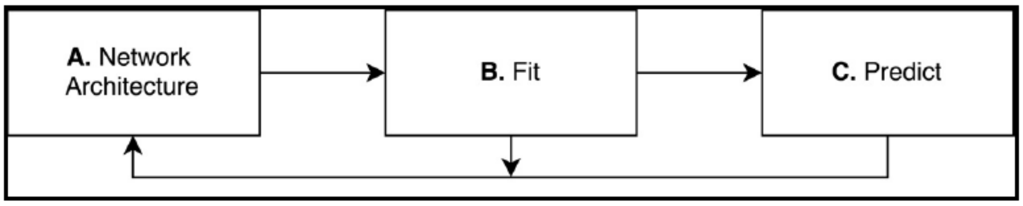


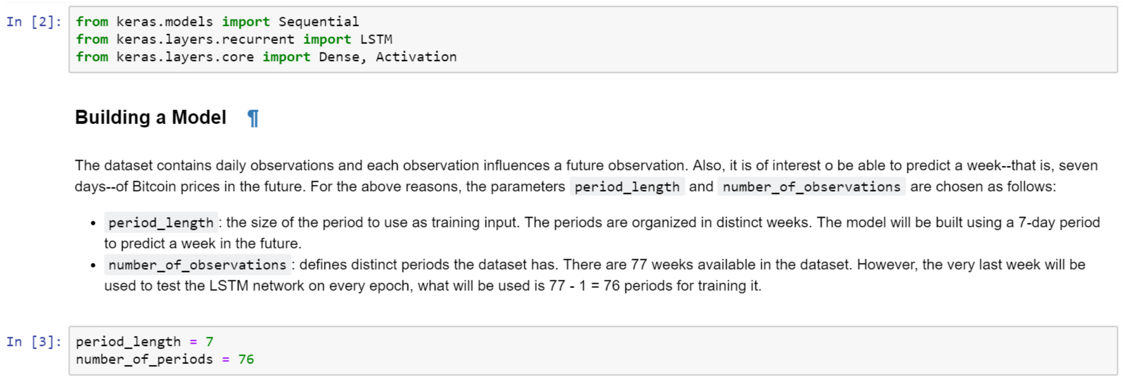
**Fig. 7**. The Keras NN paradigm: A. design a NN architecture B. Train an NN (or Fit), and C. Make predictions.

Keras allows for superior control within each of the steps in Fig 7 with a focus on making it as easy as possible for users to create neural networks in as little time as possible. That allows users to start with a simple model, then add complexity to each one of the steps above to make that initial model perform better. Avantage will be taken of this paradigm during in the current project to create the LSTM network.

### 2.2.1 TensorFlow Model with Keras

This section addresses design and compilation of a deep learning model using Keras as an interface to TensorFlow. This will be used as the base model in this project and will assist in experimenting with different optimization techniques. The essential components of the model are entirely designed in Jupyter notebook.

The first LSTM model is built parametrizing two values: the input size of the training observation (1 equivalent for a single day) and the output size for the predicted period—in of 7 days as shown in Fig. 8 below:



```
In [2]: from keras.models import Sequential
        from keras.layers.recurrent import LSTM
        from keras.layers.core import Dense, Activation
```

**Building a Model** ¶

The dataset contains daily observations and each observation influences a future observation. Also, it is of interest o be able to predict a week--that is, seven days--of Bitcoin prices in the future. For the above reasons, the parameters `period_length` and `number_of_observations` are chosen as follows:

- `period_length` : the size of the period to use as training input. The periods are organized in distinct weeks. The model will be built using a 7-day period to predict a week in the future.
- `number_of_observations` : defines distinct periods the dataset has. There are 77 weeks available in the dataset. However, the very last week will be used to test the LSTM network on every epoch, what will be used is 77 - 1 = 76 periods for training it.

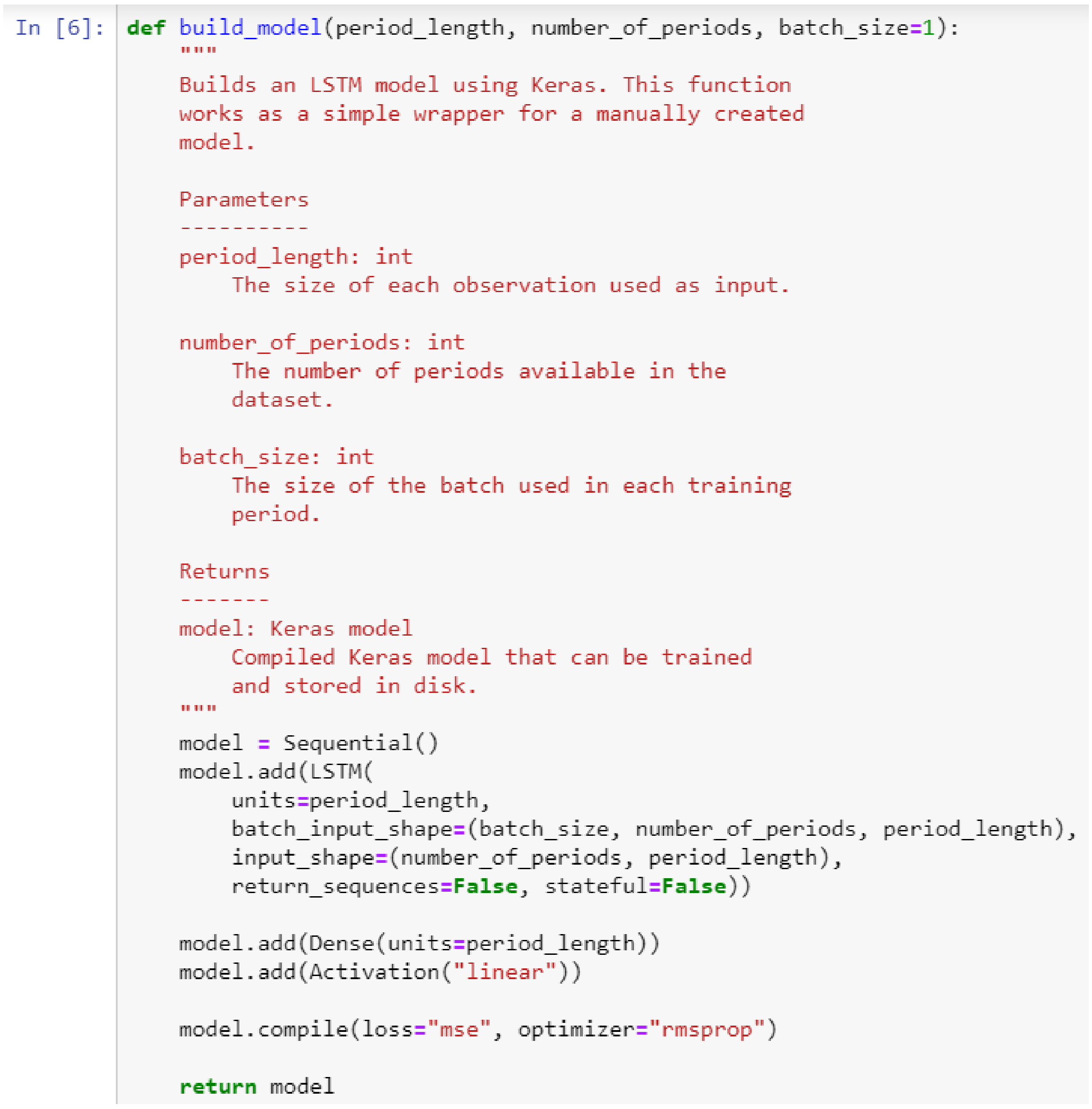
```
In [3]: period_length = 7
        number_of_periods = 76
```

**Fig. 8**. Libraries and definitions for the LSTM model

The dataset contains daily observations and each observation influences a future observation. Also, it is of interest to be able to predict a week--that is, seven days--of Bitcoin prices in the future. For the above reasons, the parameters `period_length` and `number_of_observations` are chosen as follows:

- `period_length`: the size of the period to use as training input. The periods are organized in distinct weeks. The model will be built using a 7-day period to predict a week in the future.
- `number_of_observations` : defines distinct periods the dataset has. There are 77 weeks available in the dataset. However, the very last week will be used to test the LSTM network on every epoch, what will be used is 77 - 1 = 76 periods for training it.

Next step was to build a function for the LSTM model construction. The following function in Fig. 9 builds an LSTM model using Keras and works as a simple wrapper for a manually created model.



```
In [6]: def build_model(period_length, number_of_periods, batch_size=1):
            """
            Builds an LSTM model using Keras. This function
            works as a simple wrapper for a manually created
            model.

            Parameters
            ----------
            period_length: int
                The size of each observation used as input.

            number_of_periods: int
                The number of periods available in the
                dataset.

            batch_size: int
                The size of the batch used in each training
                period.

            Returns
            -------
            model: Keras model
                Compiled Keras model that can be trained
                and stored in disk.
            """
            model = Sequential()
            model.add(LSTM(
                units=period_length,
                batch_input_shape=(batch_size, number_of_periods, period_length),
                input_shape=(number_of_periods, period_length),
                return_sequences=False, stateful=False))

            model.add(Dense(units=period_length))
            model.add(Activation("linear"))

            model.compile(loss="mse", optimizer="rmsprop")

            return model
```

**Fig. 9**. Functions for the LSTM model

Saving the model: The function `build.model()` was used as a starting point for building the model.

```
model.save('bitcoin_lstm_v0.h5')
```

The function `build_model()`is used as a starting point for building the model. That function will be refactored when building the Flask application for making it easier to train the network and use it for predictions. The model output is currently stored on disk.

However, the model `'bitcoin_lstm_v0.h5'`has not been trained yet. When a model is saved without prior training, essentially only the architecture of the model is effectively saved. The same model can be loaded again using the Keras `load_model()` function, as follows:

```
model = keras.models.load_model('bitcoin_lstm_v0.h5')
```

The steps above compile the LSTM model as TensorFlow computation graph. This model can now be trained using the training set and evaluate its results with the test set.

### 2.2.2 Data to Modelling

This section addresses the implementation aspects of a deep learning system for the Bitcoin dataset. This will be achieved by building a system that reads data from a disk and feeds it into a model as a single piece of software.

#### *2.2.2.1 Neural Network Training*

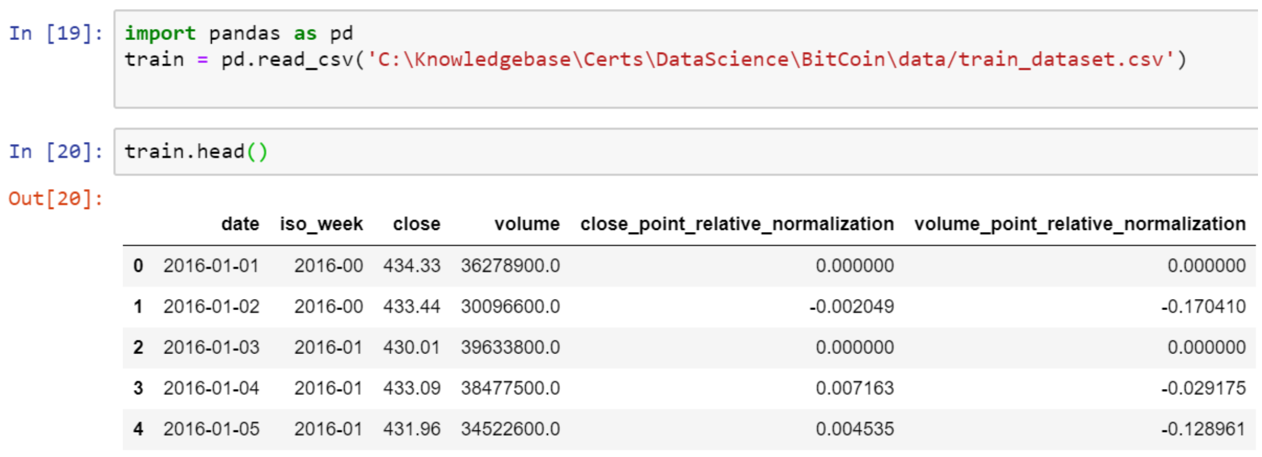
Training neural networks (NNs) can be a time consuming process. Numerous factors affect how long the training process may take. Among all the factors, following are commonly considered the most important:

- The architecture of the network
- Number of layers and neurons in the network
- Volume of data used in the training process

Several other factors may also impact how long an NN takes to train, but generally speaking, most of the optimization that a neural network can have when addressing a business problem is derived from exploring above three.

The normalized data from section 2.1, stored in train_dataset.csv, was used for this purpsoe. Following code was used to load that dataset into memory using pandas for easy exploration as shown in Fig 10. and then the pandas `head()` function was used to look at the first five rows of data:

```
import pandas as pd
train = pd.read_csv('data/train_dataset.csv')
```



```
In [19]: import pandas as pd
         train = pd.read_csv('C:\Knowledgebase\Certs\DataScience\BitCoin\data/train_dataset.csv')

In [20]: train.head()

Out[20]:
```

| | date | iso_week | close | volume | close_point_relative_normalization | volume_point_relative_normalization |
|---|---|---|---|---|---|---|
| **0** | 2016-01-01 | 2016-00 | 434.33 | 36278900.0 | 0.000000 | 0.000000 |
| **1** | 2016-01-02 | 2016-00 | 433.44 | 30096600.0 | -0.002049 | -0.170410 |
| **2** | 2016-01-03 | 2016-01 | 430.01 | 39633800.0 | 0.000000 | 0.000000 |
| **3** | 2016-01-04 | 2016-01 | 433.09 | 38477500.0 | 0.007163 | -0.029175 |
| **4** | 2016-01-05 | 2016-01 | 431.96 | 34522600.0 | 0.004535 | -0.128961 |

**Fig. 10**. Table showing the first five rows of the training dataset loaded from the `train_d-ataset.csv` file

The data from the variable `close_point_relative_normalization`, which is a normalized series of the Bitcoin closing prices—from the variable close—since the beginning of 2016.

The variable `close_point_relative_normalization` has been normalized on a weekly basis. Each observation from the week's period is made relative to the difference from the closing prices on the first day of the period. This normalization step is important and will help the network train faster.

### *2.2.2.2 Time-Series Data Reshaping*

Neural networks work with vectors and tensors, both mathematical objects that organize data in a number of dimensions. Each neural network implemented in Keras have either a vector or a tensor that is organized according to a specification as input. It can get confusing while attempting to understand how to reshape the data into the format expected by a given layer. Hence as a best practice, it is advisable to start with a network with minimum components first and then add components gradually. Keras' official documentation [3], under the section Layers, is essential for learning about the requirements for each kind of layer.

Weekly groups were created and then the resulting array rearranged to match those dimensions. The prices from the variable `close_point_relative_normalization` were organized using the weeks of both 2016 and 2017. Distinct groups containing seven observations each (one for each day of the week) were constructed for a total of 77 complete weeks.

This was done because prediction of the prices of a week's worth of trading is of interest. The ISO standard was used to determine the beginning and the end of a week; whereas kinds of organizations are entirely possible, ISO is simple and intuitive to follow, but there is room for improvement.

LSTM networks require vectors with three dimensions. These dimensions are:

- **Period length**: The period length, i.e. how many observations is there on a period.
- **Number of periods**: How many periods are available in the dataset.
- **Number of features**: Number of features available in the dataset.

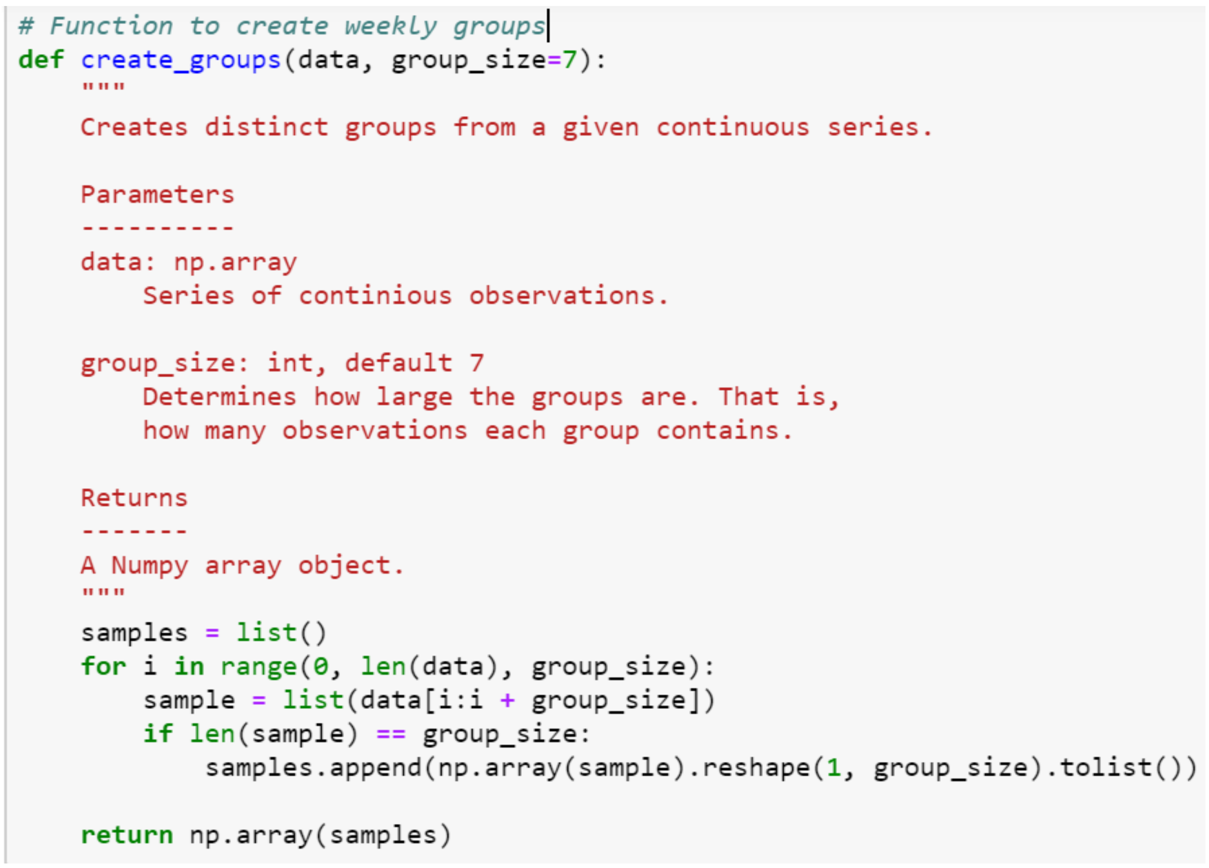
The data from the variable `close_point_relative_normalization`, as stored and viewed at this point, is a one-dimensional vector which needs to be reshaped to match the three dimensions.

For a week's period, the period length is seven days (period length = 7). We have 77 complete weeks available in our data. The very last of those weeks was used to test the model during its training period. That leaves 76 distinct weeks (number of periods = 76). Finally, a single feature will be sued in this network (number of features = 1).

In order to reshape the data to match those dimensions a combination of base Python properties and the reshape() from the numpy library was used. The 76 distinct week groups were created with seven days each using pure Python as follows:

```
group_size = 7
samples = list()
for i in range(0, len(data), group_size):
sample = list(data[i:i + group_size])
if len(sample) == group_size:
samples.append(np.array(sample).reshape(group_size,1).tolist())
data = np.array(samples)
```

A function `create_groups` was created for this purpose as follows:



```
# Function to create weekly groups
def create_groups(data, group_size=7):
    """
    Creates distinct groups from a given continuous series.

    Parameters
    ----------
    data: np.array
        Series of continious observations.

    group_size: int, default 7
        Determines how large the groups are. That is,
        how many observations each group contains.

    Returns
    -------
    A Numpy array object.
    """
    samples = list()
    for i in range(0, len(data), group_size):
        sample = list(data[i:i + group_size])
        if len(sample) == group_size:
            samples.append(np.array(sample).reshape(1, group_size).tolist())

    return np.array(samples)
```

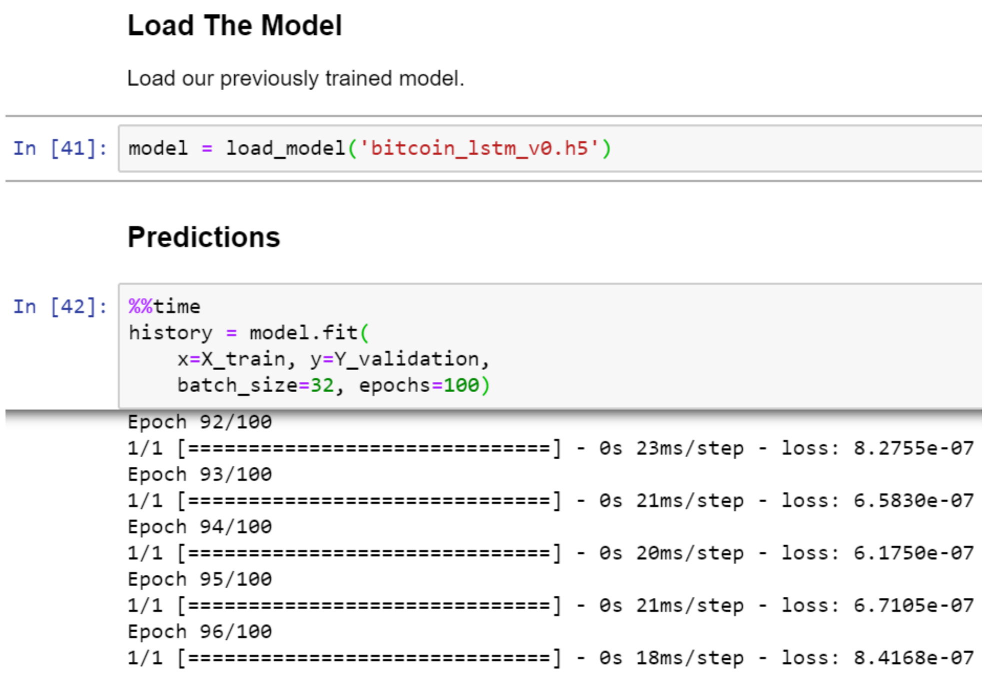
The resulting data is a variable that contains all the right dimensions. The Keras LSTM layer expects these dimensions to be organized in a specific order: number of features, number of observations, and period length. Hence reshaping of data is important.

The dataset was separated and reshaped using numpy `reshape()` to match the LSTM format as follows:

```
X_train = data[:-1,:].reshape(1, 76, 7)
Y_validation = data[-1].reshape(1, 7)
```

Each Keras layer expects its input to be organized in specific ways. However, Keras will reshape data accordingly in most cases. Reference to the Keras documentation on layers (https://keras.io/layers/core/ ) is strongly recommnded before addition of a new layer or if issues are encountered with the shape that layers expect.

Once the data is reshaped, it was ready for training. The previously saved model was loaded and trained with a given number of 100 epochs in batches of 32 as shown in Fig.11.



**Load The Model**

Load our previously trained model.

```
In [41]: model = load_model('bitcoin_lstm_v0.h5')
```

**Predictions**

```
In [42]: %%time
         history = model.fit(
             x=X_train, y=Y_validation,
             batch_size=32, epochs=100)
```

```
Epoch 92/100
1/1 [==============================] - 0s 23ms/step - loss: 8.2755e-07
Epoch 93/100
1/1 [==============================] - 0s 21ms/step - loss: 6.5830e-07
Epoch 94/100
1/1 [==============================] - 0s 20ms/step - loss: 6.1750e-07
Epoch 95/100
1/1 [==============================] - 0s 21ms/step - loss: 6.7105e-07
Epoch 96/100
1/1 [==============================] - 0s 18ms/step - loss: 8.4168e-07
```

**Fig. 11**. Loading of the earlier model and training it with new data

In Fig. 11 the logs of the model is stored in a variable called `history`. The model logs are useful for exploring specific variations in its training accuracy and to understand how well the loss function is performing.

LSTMs are computationally expensive models. They may take up to file minutes to train with our dataset in a modern computer. Most of that time is spent at the beginning of the computation, when the algorithm creates the full computation graph. The process gains speed after it starts training as shown in Fig 12:

```
In [44]: pd.Series(history.history['loss']).plot(linewidth=2, figsize=(14, 4), color='#d35400')

Out[44]: <matplotlib.axes._subplots.AxesSubplot at 0x1fa70c8f1d0>
```

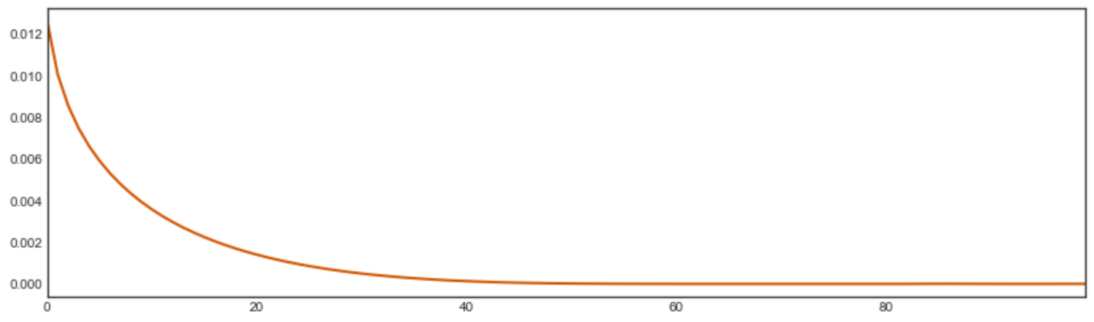


**Fig. 12**. Plot for the loss function evaluated at each epoch of the 100 epochs defined

Fig. 12 compares prediction of the model at each epoch, then compares with the real data using a technique called mean-squared error.

At a glance, the network appears to perform well as it starts with a very small error rate which decreases continuously.

### 2.2.3 Predictions

At this stage, once the network was trained with new data,  the predictive model was ready to be tested. Once the training was done with model.fit(), the predictions became trivial:

```
model.predict(x=X_train)
```

The data for making predictions was the same as the data used for training (the `X_train` variable). If more data becomes available, that can also be used instead—given that it is reshaped to the format the LSTM requires.

The model returned a list of normalized values with the prediction for the next seven days (Fig. 13). The denormalize() function was sued to turn the data into US Dollar values. The latest values available was used as a reference for scaling the predicted results:

```
denormalized_prediction = denormalize(predictions, last_weeks_value)
```

```
predictions = model.predict(x=X_train)[0]
```

```
last_weeks_value = train[train['date'] == train['date'].max()]['close'].values[0]
denormalized_prediction = denormalize(predictions, last_weeks_value)
```

```
pd.DataFrame(denormalized_prediction).plot(linewidth=2, figsize=(6, 4), color='#d35400', grid=True)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x1fa71d23710>
```

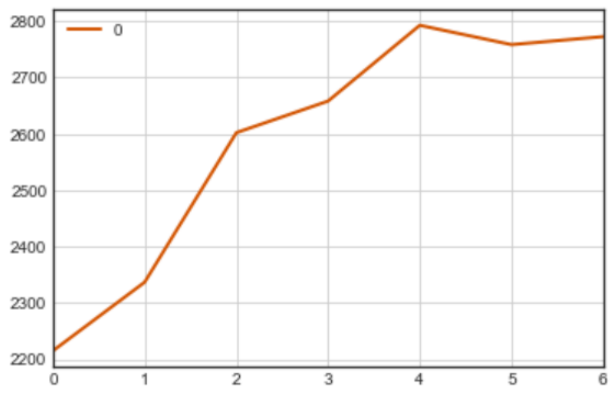


**Fig. 13**. Post de-normalization LSTM model prediction

Fig. 13 shows after de-normalization, our LSTM model predicted that in late July 2017, the prices of Bitcoin would increase from $2,200 to roughly $2,800, a 30 percent increase in a single week. The predictions suggest a price surge of approximately 30% over next 7 days.

```
full_series = list(train['close'].values) + list(denormalized_prediction)
pd.DataFrame(full_series[-7*8:]).plot(linewidth=2, figsize=(14, 4), color='#d35400', grid=True)
plt.axvline(len(full_series[-7*8:]) - 7)
```

```
<matplotlib.lines.Line2D at 0x1fa71e84358>
```

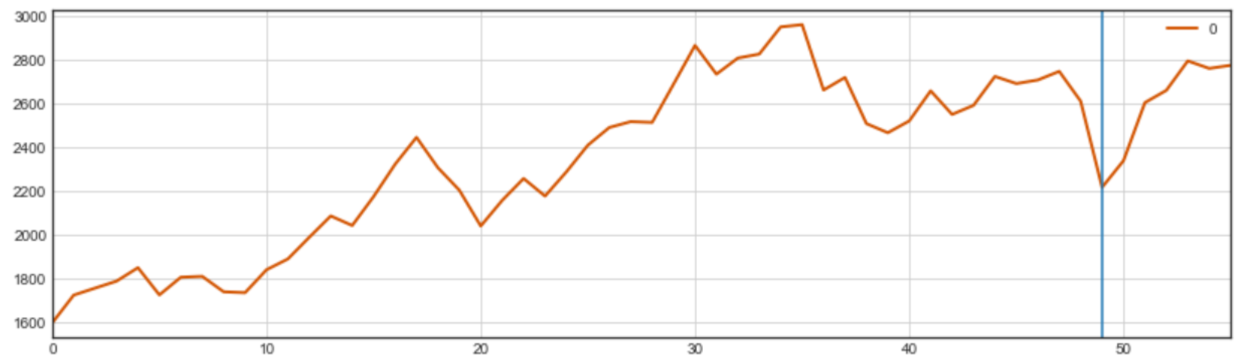


**Fig. 14**. Projection of Bitcoin prices for seven days in the future using the LSTM model

Fig. 14 above shows the combination of both time-series in this graph: the real data (before the blue line) and the predicted data (after the blue line). The model shows variance similar to the patterns seen before and it suggests a price increase during the following seven days period.

### 2.2.4 Conclusion

In this section saw a complete deep learning system assembled: from data to prediction. The model created in this activity need a number of improvements before it can be considered useful. However, it serves as a solid starting point from which it can continuously improve. LSTMs also suffer from overfitting, a phenomenon common in neural networks in which they learn patterns from the training data that are useless when predicting real-world patterns.

## 2.3 Model Evaluation, Optimization & Hyperparameter Tuning

This section focusses on the evaluation of the neural network (NN) model.

### 2.3.1 Bitcoin Model Evaluation

For the Bitcoin model an LSTM architecture has been used. LSTMs are designed to predict sequences. Because of that, a set of variables to predict a different single variable is not of use here — even though this is a regression problem. Instead, previous observations from a single variable (or set of variables) have been used to predict future observations of that same variable (or set). The y parameter on `Keras.fit()` contains the same variable as the x parameter, but only the predicted sequences.

The Bitcoin test set used has 19 weeks of Bitcoin daily price observations, which is equivalent to about 20 percent of the original dataset. The neural network has also been trained using the other 80 percent of data - i.e., the train set with 56 weeks of data, minus one for the validation set - and the trained network stored on disk (`bitcoin_lstm_v0`). Now, the `evaluate()` method is used in each one of the 19 weeks of data from the test set and investigated for neural network performance. This performance measurement will involve the total of 76 preceding weeks because the network has been trained to predict one week of data using exactly 76 weeks of continuous data. This is an important aspect from *model productization* point of view.

```
        evaluated_weeks = []
        for i in range(0, test_data.shape[1]):
```

```
input_series = combined_set[0:,i:i+77]

X_test = input_series[0:,:-1].reshape(1, input_series.shape[1]-
1, 7)
Y_test = input_series[0:,-1:][0]

result = model.evaluate(x=X_test, y=Y_test, verbose=0)
evaluated_weeks.append(result)
```

In the above code snippet (shown in Jupyter notebook for this section), each week was evaluated using Keras' `model.evaluate()` , then its output stored in the variable evaluated weeks. The following plot depicts the resulting MSE for each week in Fig. 15:

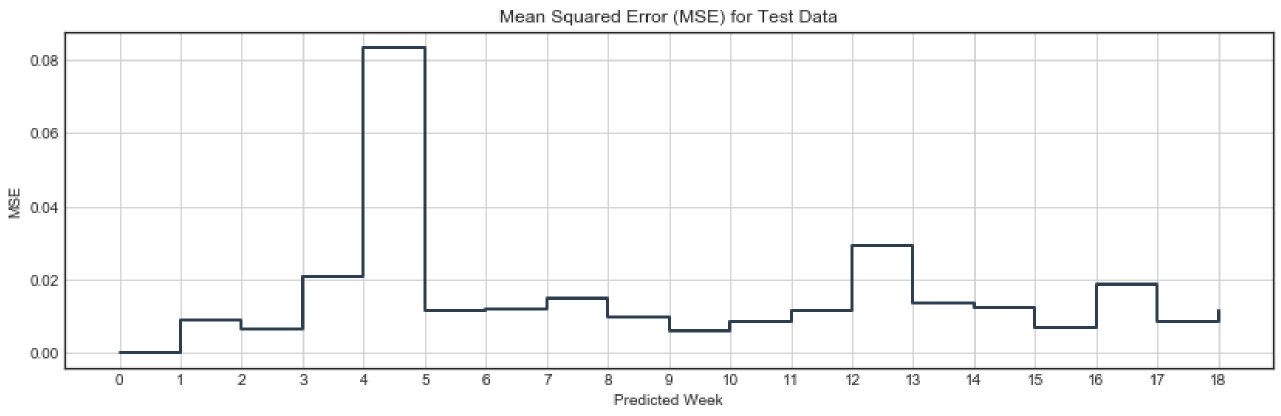


**Fig. 15**. MSE for each week in the test set. During week 5, the model predictions are worse than in other weeks

The MSE in Fig. 15 suggests that the model performs well during most weeks, except for week 5, when its value increases to about 0.08. The model seems to be performing well for almost all of the other test weeks.

### 2.3.2 Overfitting

It is important to look into the first trained network (bitcoin_lstm_v0) for signs of overfitting. Overfitting happens when a model is trained to optimize a validation set, but it does so at the expense of more generalizable patterns from the phenomenon of interest. The main challenge with overfitting is that a model learns how to predict the validation set, but fails to predict efficiently for new data.

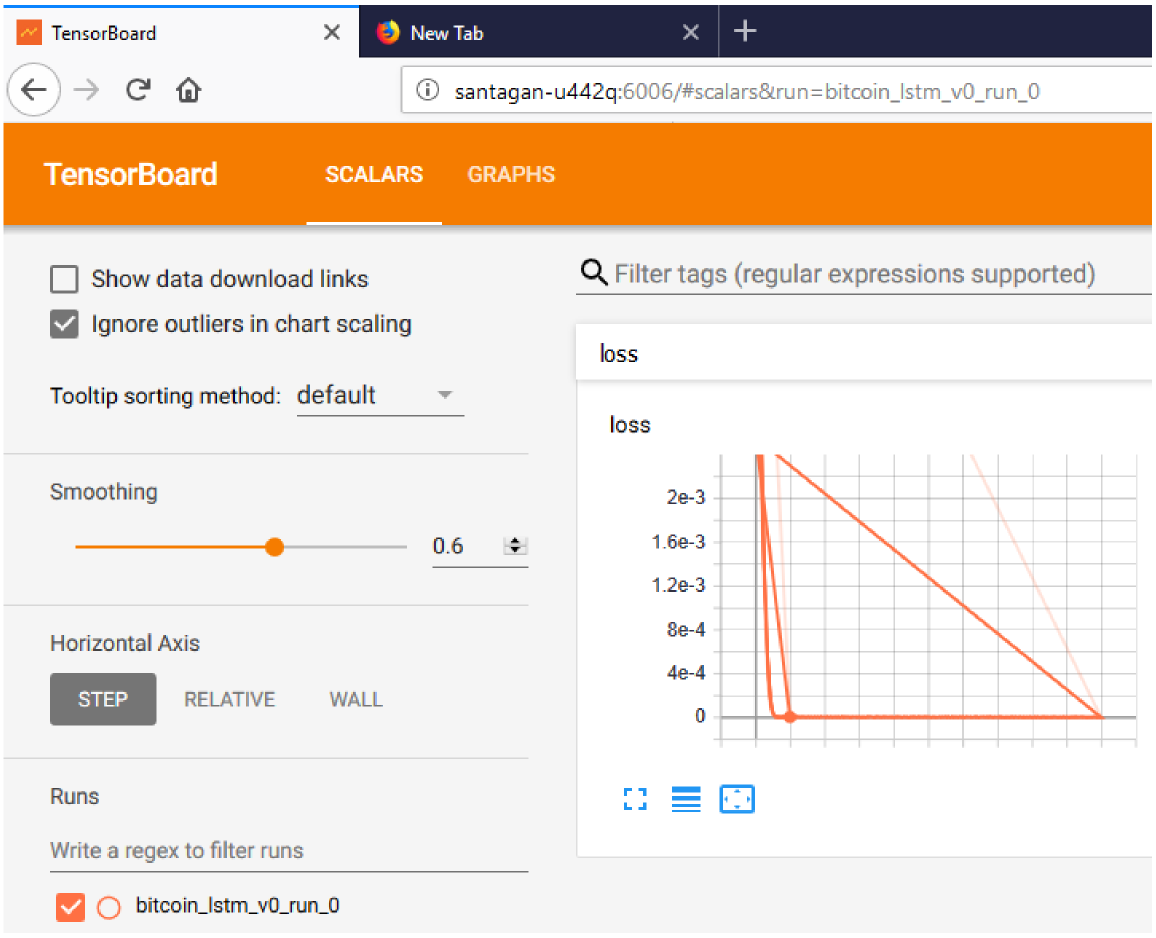
The loss function used in the current model reaches very low levels (about 2.9 * 10-6) at the end of the training process. This happens early: the MSE loss function used to predict the last week in the data decreases to a stable plateau at about epoch 30. This indicates that the model is predicting the data from week 77 almost perfectly, using the preceding 76 weeks. It needs investigation to deduce whether this is a result of overfiting.

In Fig 15 the LSTM model reaches extremely low values in the validation set (about 2.9 * 10-6), yet it also reaches low values in the test set. The key difference, is in the scale. The MSE for each week in the test set is about 4,000 times bigger (on average) than in the test set. This infers that the model is performing much worse in the test data than in the validation set. This is worth looking into. The scale, though, hides the power of the LSTM model: even performing much worse in the test set, the predictions' MSE errors are still very, very low. That suggests that our model may be learning patterns from the data.

### 2.3.3 Model Predictions

To perform efficient model predictions, TensorBoard was started as follows and as shown in Fig. 16:

```
(qml) C:\Knowledgebase\Certs\DataScience\Projects>tensorboard --
logdir=logs/

W0624 08:15:45.804615 24008 plugin_event_accumulator.py:294] Found
more than one graph event per run, or there was a metagraph
containing a graph_def, as well as one or more graph events.
Overwriting the graph with the newest event.

W0624 08:15:45.807575 24008 plugin_event_accumulator.py:302] Found
more than one metagraph event per run. Overwriting the metagraph
with the newest event.

TensorBoard 1.13.1 at http://SANTAGAN-U442Q:6006 (Press CTRL+C to
quit)
```


santagan-u442q:6006/#scalars&run=bitcoin_lstm_v0_run_0

Show data download links: ☐
Ignore outliers in chart scaling: ✓
Tooltip sorting method: default
Smoothing: 0.6
Horizontal Axis: STEP
Runs: bitcoin_lstm_v0_run_0 ✓

**Fig. 16**. TensorBoard Interface

In order to perform the model predictions, an active environment was created in JUpyter Notebook (please see attached codes or Appendix for this section):

```
import math
import numpy as np
import pandas as pd
import seaborn as sb
import matplotlib.pyplot as plt
```

```
from keras.models import load_model
from keras.callbacks import TensorBoard
from datetime import datetime, timedelta
```

```
Using TensorFlow backend.
```

```
from utilities import create_groups, split_lstm_input
```

```
plt.style.use('seaborn-white')
```

In addition to comparing model MSE errors, it is also important to be able to interpret its results intuitively. Using the same model, a series of predictions were created for the following weeks, using 76 weeks as input. This was done by sliding a window of 76 weeks over the complete series (that is, train plus test sets), and making predictions for each of those windows. Predictions were done using the `Keras model.predict()` method:

```
combined_set = np.concatenate((train_data, test_data), axis=1)
        predicted_weeks = []
        for i in range(0, validation_data.shape[1] + 1):
        input_series = combined_set[0:,i:i+76]
        predicted_weeks.append(B.predict(input_series))
```

The above code shows predictions using model.predict(), then store these predictions in the predicted_weeks variable. The plot of the resulting predictions were generated as shown in the following Fig 17:

```
plot_two_series(observed, predicted,
                variable='close_point_relative_normalization',
                title='Normalized Predictions per Week')
```

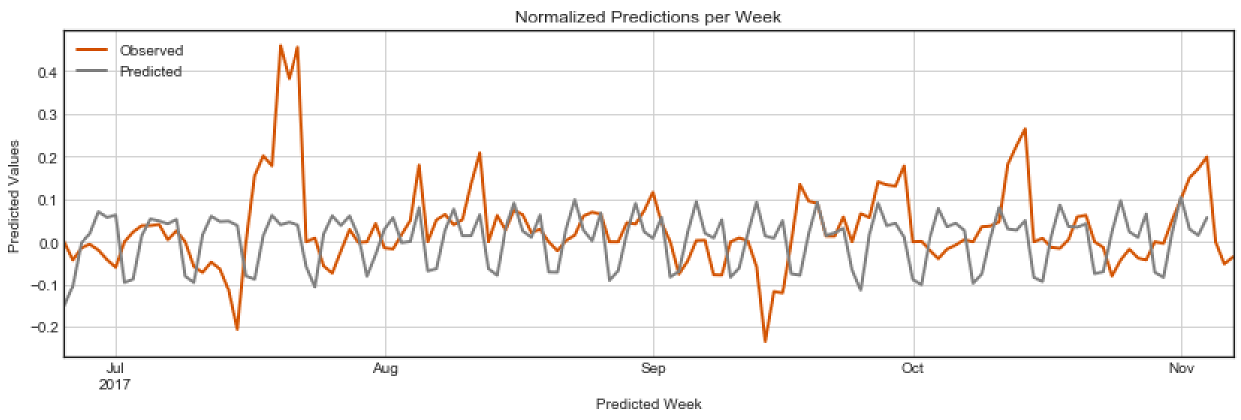


**Fig. 17**. MSE for each week in the test set. In week 5, the model predictions are worse than in any other week.

The results of our model in Fig. 17 suggest that its performance is not too bad. By observing the pattern from the Predicted line, it was noticed that the network had identifiled a fluctuating pattern occuring on a weekly basis, in which the normalized prices go up in the middle of the week, then down by the end of it. With the exception of a few weeks — most notably week 5 — most weeks fall close to the correct values.

The predictions were now denormalized so that the prediction values could be investigated using the same scale as the original data in US Dollars. This was done by implementing a denormalization function that uses the day index from the predicted data to identify the equivalent week on the test data. After that week is identified, the function then takes the first value of that week and uses that value to denormalize the predicted values by using the same point-relative normalization technique, but inverted:

```
def denormalize(reference, series,
normalized_variable='close_point_relative_normalization',
denormalized_variable='close'):

      week_values =
      observed[reference['iso_week']==series['iso_week'].
      values[0]]
      last_value = week_values[denormalized_variable].values[0]
      series[denormalized_variable] =
      last_value*(series[normalized_variable]+1)
      return series

predicted_close = predicted.groupby('iso_week').apply
      (lambda x: denormalize(observed, x))
```

Fig 18. Below shows the de-normalized plot:

```
plot_two_series(observed, predicted_close,
                variable='close',
                title='De-normalized Predictions per Week')
```

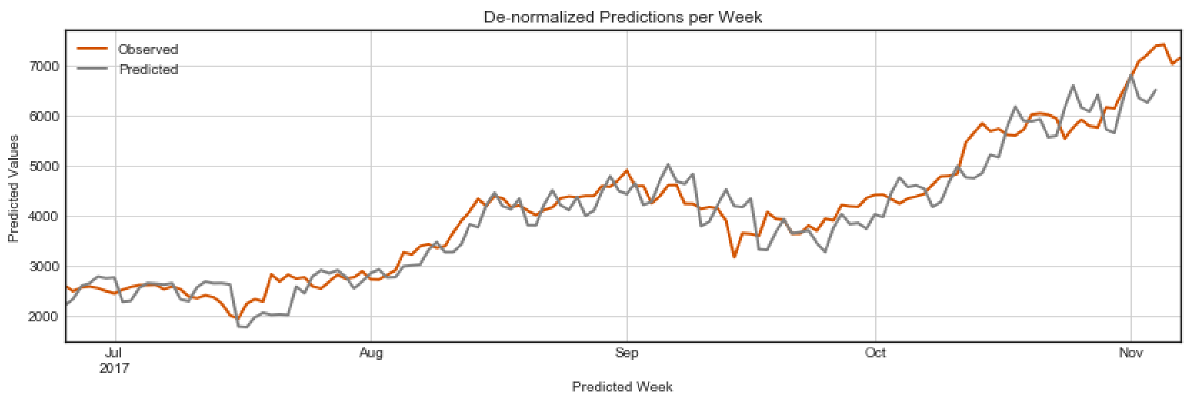


**Fig. 18**. De-normalized plot for the model predictions.

The results now compare the predicted values with the test set, using US Dollars. As seen previously in Figure 15, the `bitcoin_lstm_v0` model seems to perform quite well in predicting the Bitcoin prices for the following seven days.

### 2.3.4 Interpreting Predictions

Even though the `bitcoin_lstm_v0` model seems to perform quite well in predicting the Bitcoin prices for the following seven days, it is of importance to find out if the performance is measurable in interpretable terms.

Fig. 15 and 17 appears to depict that the model prediction matches the test data closely. Keras' `model.evaluate()` function is useful for understanding how a model is performing at each evaluation step. However, given normalized datasets are being used to train neural networks, the metrics generated by the `model.evaluate()` method are also hard to interpret.

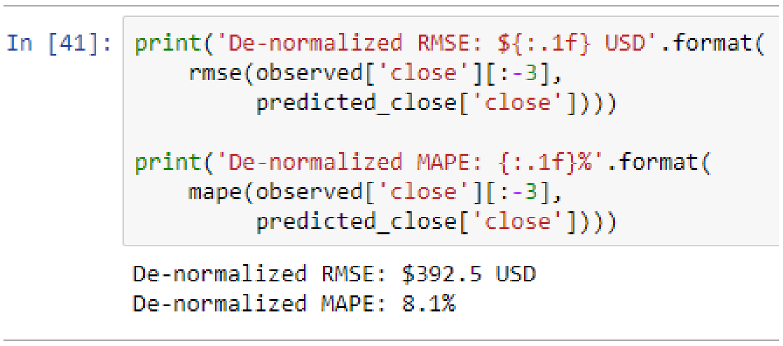
To address that issue, the complete set of predictions were collected from the model and compared with the test set using two other functions from **Table 1** in **Section 1.6.2** which are easier to interpret: MAPE and RMSE, implemented as `mape()` and `rmse()` , respectively, implemented using numpy [7]:

```
def mape(A, B):
return np.mean(np.abs((A - B)/A))*100

def rmse(A, B):
return np.sqrt(np.square(np.subtract(A, B)).mean())
```

After comparing the test set with the predictions using both of those functions, following results were obtained as shown in Fig. 19:

- Denormalized RMSE: **$392.5**
- Denormalized MAPE: **8.1 percent**

The values indicate that the predictions differ, on average, about $392.5 from real data. That represents a difference of about 8.1% from real Bitcoin prices.



```
In [41]: print('De-normalized RMSE: ${:.1f} USD'.format(
             rmse(observed['close'][:-3],
                  predicted_close['close'])))

         print('De-normalized MAPE: {:.1f}%'.format(
             mape(observed['close'][:-3],
                  predicted_close['close'])))

         De-normalized RMSE: $392.5 USD
         De-normalized MAPE: 8.1%
```

**Fig. 19**. MSE for each week in the

These results facilitate the understanding of the predictions. The model.evaluate() method will be further used to keep track of how the LSTM model performance. In addition, both rmse() and mape() will be computed on the complete series on every version of the model to interpret how close the model is to predict Bitcoin prices.

### 2.3.5 Conclusion:

This section stated an evaluation scheme of a network using loss functions. Loss functions are key elements of neural networks as they evaluate the performance of a network at each epoch and are the starting point for the propagation of adjustments back into layers and nodes. In addition, causes of some loss functions were explored as it can be difficult to interpret (for instance, the MSE) and developed a strategy using two other functions — RMSE and MAPE—to interpret the predicted results from our LSTM model. Most importantly, this section established an active training environment, a system that can train a deep learning model and evaluate its results continuously.

### 2.3.6 Hyperparameter Optimization

This section will address the common strategies for improving the performance of neural network models:

- Adding or removing layers and changing the number of nodes
- Increasing or decreasing the number of training epochs
- Experimenting with different activation functions
- Using different regularization strategies

Neural networks with single hidden layers can perform fairly well on many problems. The fist Bitcoin model (`bitcoin_lstm_v0`) is a good example of this: it can predict the next 7 days of Bitcoin prices (from the test set) with error rates of about 8.1 percent using a single LSTM layer. However, not all problems can be modelled with single layers. The more complex the function that is being worked upon is, the higher the likelihood that more added layers will be needed. Typically, each layer creates a model representation of its input data. Earlier layers in the chain create lower-level representations, and later layers, higher-level.

The number of neurons that a layer requires is related to how both the input and output data are structured [8]. Generally new neurons are added alongside the addition of new layers. Then, one can add a layer that has either the same number of neurons as the previous one, or a multiple of the number of neurons from the previous layer. For example, if the fist hidden layer has 12 neurons, then one can experiment with adding a second layer that has either 12, 6, or 24. Adding layers and neurons can have significant performance limitations. It is common to start with a smaller network (that is, a network with a small number of layers and neurons), then grow according to its performance gains.

One of the factors to be aware of is the following: the more added layers, the more hyperparameters will be needed to be tuned and the longer the network will take to train. If the model is performing fairly well and not overfitting the data, then experimentation with the other strategies before adding new layers to the network is recommended.

The original LSTM model was modified by adding more layers. In LSTM models, one typically adds LSTM layers in a sequence, making a chain between LSTM layers. In this case, the new LSTM layer had the same number of neurons as the original layer, hence there was no need to configure that parameter. The modified version of the model was named `bitcoin_lstm_v1`. It is deemed a good practice to name each one of the models in which one is attempting different hyperparameter configurations differently. This helps to keep track of how each different architecture performs, and also to easily compare model differences in TensorBoard.

Prior to adding a new LSTM layer, the parameter `return_sequences` need to be changed to *True* on the first LSTM layer. The reason for this is the first layer expects a sequence of data with the same input as that of the first layer. When this parameter is set to *False*, the LSTM layer outputs the predicted parameters in a different, incompatible output.

```
period_length = 7
number_of_periods = 76
batch_size = 1

model = Sequential()
model.add(LSTM(
     units=period_length,
      batch_input_shape=(batch_size, number_of_periods,
      period_length),  input_shape=(number_of_periods,
      period_length),
      return_sequences=True, stateful=False))

model.add(LSTM(
      units=period_length,
      batch_input_shape=(batch_size, number_of_periods,
      period_length),
      input_shape=(number_of_periods, period_length),
return_sequences=False, stateful=False))

model.add(Dense(units=period_length))
model.add(Activation("linear"))
model.compile(loss="mse", optimizer="rmsprop")
```

#### *2.3.6.1 Epochs*

Epochs are the number of times the network adjust its weights in response to data passing through and its loss function. Running a model for more epochs can allow it to learn more from data, but may also introduce the risk of overfitting. When training a model, general preferance is to increase the epochs exponentially until the loss function starts to plateau. In the case of the bitcoin_lstm_v0 model, its loss function plateaus at about 100 epochs.

The current LSTM model uses a small amount of data to train, so increasing the number of epochs is not expected to affect its performance in significant ways. For instance, if one attempts to train it at 103 epochs, the model barely gains any improvements. This will not be the case if the model being trained uses enormous amounts of data. In those cases, a large number of epochs is crucial to achieve good performance. As a rule of thumb, the larger the data used to train the model, the more epochs it will need to achieve good performance.

Due to the small size of the Bitcoin dataset, increasing the epochs that the model trains may have only a marginal effect on its performance. In order to have the model train for more epochs, one only has to change the epochs parameter in `model.fit()` :

```
number_of_epochs = 10**3
model.fit(x=X, y=Y, batch_size=1,
        epochs=number_of_epochs,
        verbose=0,
callbacks=[tensorboard])
```

#### *2.3.6.2 Activation Function – Implementation*

Activation functions were implemented in Keras by instantiating the `Activation()` class and adding it to the `Sequential()` model. The `tanh` function was used to achieve this.

After implementing the activation function, the version of the model was upgraded to `bitcoin_lstm_v3`:

```
model = Sequential()
model.add(LSTM(
        units=period_length,
        batch_input_shape=(batch_size, number_of_periods,
        period_length),
        input_shape=(number_of_periods, period_length),
        return_sequences=True, stateful=False))
model.add(LSTM(
        units=period_length,
        batch_input_shape=(batch_size, number_of_periods,
        period_length),
        input_shape=(number_of_periods, period_length),
        return_sequences=False, stateful=False))
model.add(Dense(units=period_length))
model.add(Activation("tanh"))
model.compile(loss="mse", optimizer="rmsprop")
```

### *2.3.6.3 Regularization*

As discussed before, neural networks are particularly prone to overfitting. Regularization strategies refer to techniques that deal with the problem of overfitting by adjusting how the network learns.

**Dropout** is a regularization technique based on a simple question: if one randomly takes away a proportion of nodes from layers, how will the other node adapt? It turns out that the remaining neurons adapt, learning to represent patterns that were previously handled by those neurons that are missing.

The dropout strategy is simple to implement and is typically very effective to avoid overfitting. This was the preferred regularization strategy for this part of the project.

In order to implement the dropout strategy using Keras, the `Dropout()` class was imported and added to the network immediately after each LSTM layer.

This addition effectively makes the network go up by another version to `bitcoin_lstm_v4`:

```
    model = Sequential()
    model.add(LSTM(
            units=period_length,
            batch_input_shape=(batch_size, number_of_periods, period_length),
            input_shape=(number_of_periods, period_length),
            return_sequences=True, stateful=False))

    model.add(Dropout(0.2))
    model.add(LSTM(
            units=period_length,
            batch_input_shape=(batch_size, number_of_periods, period_length),
            input_shape=(number_of_periods, period_length),
            return_sequences=False, stateful=False))

    model.add(Dropout(0.2))
    model.add(Dense(units=period_length))
    model.add(Activation("tanh"))

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

### *2.3.6.4 Optimization Results*

A total of four versions of the model. In order to evaluate which model performs best the same metrics were used as in the first model: MSE, RMSE, and MAPE. MSE is used to compare the error rates of the model on each predicted week. RMSE and MAPE are computed to make the model results easier to interpret. Table 3 shows the comparisons.

**Table 3 Optimization Results for All Models**

| Model | MSE (last epoch) | RMSE ( entire series) | MAPE (entire series) | Training Time |
|---|---|---|---|---|
| bitcoin_lstm_v0 | | 392.5 | 8.10% | |
| bitcoin_lstm_v1 | $6.55*10^{-6}$ | 410.2 | 8.30% | 31.3 s |
| bitcoin_lstm_v2 | $3.42*10^{-6}$ | 419.6 | 8.70% | 48.2 s |
| bitcoin_lstm_v3 | $2.7*10^{-4}$ | 417.3 | 8.50% | 58.1 s |
| bitcoin_lstm_v4 | $4.8*10^{-4}$ | 439.8 | 8.40% | 73s |

From Table 3, it appears that the fist model (`bitcoin_lstm_v0`) performed the best in nearly all defined metrics.

### 2.3.7 Conclusion

The model was evaluated using the metrics mean squared error (MSE), squared mean squared error (RMSE), and mean averaged percentage error (MAPE). The latter was computed using two metrics in a series of 19 week predictions made by the first neural network model from which proved to be performing well. The same model was also optimized using optimization techniques typically used to increase the performance of neural networks. A number of these techniques were implemented to predict Bitcoin prices with different error rates.

# 3 The Wisconsin Breast Cancer Dataset

The dataset for this part of the Project is taken from the publicly avaialble Wisconsin Breast Cancer database, created by Dr. William H. Wolberg, a physician at the University of Wisconsin Hospital at Madison, Wisconsin, USA. This dataset contains 569 samples of malignant and benign tumor cells. Dr. Wolberg used fluid samples, taken from patients with solid breast masses and an easy-to-use graphical computer program called Xcyt, which is capable of performing the analysis of cytological features based on a digital scan. The program uses a curve-fitting algorithm to compute ten features from each one of the cells in the sample and then calculates the mean value, extreme value and standard error of each feature for the image.

The first two columns in the dataset store the unique ID numbers of the samples and the corresponding diagnoses (M = malignant, B = benign), respectively. Columns 3-32 contain 30 real-valued features that have been computed from digitized images of the cell nuclei, which can be used to build a model to predict whether a tumor is benign or malignant.

The Breast Cancer Wisconsin dataset resides in the UCI Machine Learning Repository, and more information about this dataset can be found at
https://archive.ics.uci.edu/ml/datasets/Breast+Cancer+Wisconsin+(Diagnostic)

Breast cancer (BC) is one of the most common cancers among women worldwide, representing the majority of new cancer cases and cancer-related deaths according to global statistics, making it a significant public health problem in today’s society.

The early diagnosis of BC can improve the prognosis and chance of survival significantly, as it can promote timely clinical treatment to patients. Further accurate classification of benign tumors can prevent patients undergoing unnecessary treatments. Thus, the correct diagnosis of BC and classification of patients into malignant or benign groups is the subject of much research. Because of its unique advantages in critical features detection from complex BC datasets, machine learning (ML) is widely recognized as the methodology of choice in BC pattern classification and forecast modelling.

## 3.1 Objective

This work aims to look at visualisation options which are most helpful in treating such data. Classification and data mining methods are an effective way to classify data. Especially in medical field, where those methods are widely used in diagnosis and analysis to make decisions. This work aims to analyse which features are most helpful in predicting malignant or benign cancer and to investigate general trends that may aid in model selection and hyper parameter selection. To achieve this, machine learning classification methods have been used to fit functions that can predict the discrete class of new input.

**Supervised Learning**: Supervised learning is used whenever it becomes necessary to predict a certain outcome from a given input, and examples of input/output pairs are available as labelled data. A machine learning model is built from these input/output pairs, which comprise the training set. The goal is to make accurate predictions for new, never-before-seen data. Supervised learning often requires human effort to build the training set, but afterward automates and often speeds up an otherwise laborious or infeasible task.

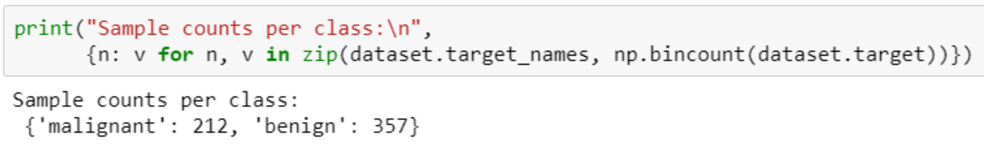
## 3.2 Data Exploration Methods & Software

Table 1. below shows software components were used for this Project:

**Table 1. Software Used**

| Software | Description | Version |
|---|---|---|
| Python | Programming language | 3.6 |
| Jupyter Notebook | Browser-based software for working interactively with Python sessions. | |
| Pandas | Python package for analyzing and manipulating data. | |
| Numpy | Python package for high-performance numerical computations. | |

Initially, the data was loaded and the raw data was inspected in csv format and `key()` `target_names` and `target` were inspected for counts of malignant and benign (Fig. 1)



```
print("Sample counts per class:\n",
      {n: v for n, v in zip(dataset.target_names, np.bincount(dataset.target))})
```

```
Sample counts per class:
 {'malignant': 212, 'benign': 357}
```

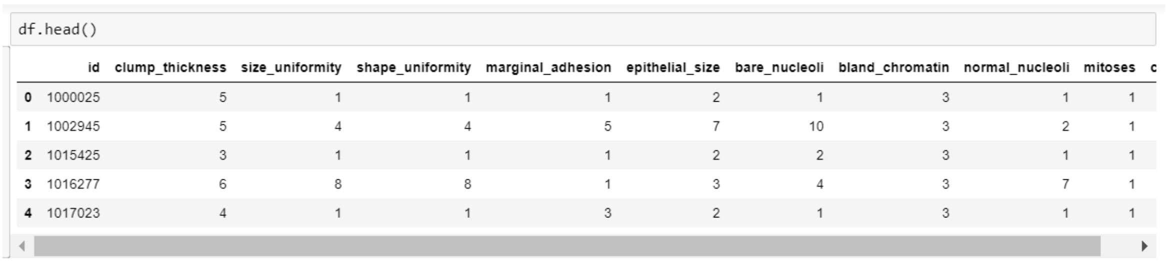
**Fig. 1** Malignant and benign counts

From the above it could be concluded that out of the 569 persons, 357 are labeled as B (benign) and 212 as M (malignant).

### 3.2.1 Data Clean up

Data cleaning techniques addresses handling missing values, specifies the use of blanks, question marks, special characters, or other strings that may indicate the null condition (where a value for a given attribute is not available), and how such values should be handled. The null rule should specify how to record the null condition, for example, such as to store zero for numerical attributes, a blank for character attributes, or any other conventions that may be in use.

In order to inspect the headers, the python `head()` function was used to look at the first five rows of the data as shown in Fig 2 below



```
df.head()
```

| | id | clump_thickness | size_uniformity | shape_uniformity | marginal_adhesion | epithelial_size | bare_nucleoli | bland_chromatin | normal_nucleoli | mitoses | c |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **0** | 1000025 | 5 | 1 | 1 | 1 | 2 | 1 | 3 | 1 | 1 | |
| **1** | 1002945 | 5 | 4 | 4 | 5 | 7 | 10 | 3 | 2 | 1 | |
| **2** | 1015425 | 3 | 1 | 1 | 1 | 2 | 2 | 3 | 1 | 1 | |
| **3** | 1016277 | 6 | 8 | 8 | 1 | 3 | 4 | 3 | 7 | 1 | |
| **4** | 1017023 | 4 | 1 | 1 | 3 | 2 | 1 | 3 | 1 | 1 | |

**Fig. 2** First five rows

Then the data was inspected for nulls as shown in Fig. 3

```
## Handling NA

df = df.drop(['bare_nucleoli'], axis=1)
df.isnull().sum()
```

```
id                  0
clump_thickness     0
size_uniformity     0
shape_uniformity    0
marginal_adhesion   0
epithelial_size     0
bland_chromatin     0
normal_nucleoli     0
mitoses             0
class               0
dtype: int64
```

**Fig. 3** Finding nulls

## 3.2.2 Data Visualization with Heat Map

A Heat Map was generated in order to understand the correlation between various attributes (parameter) as shown in Fig. 4.

```
import seaborn as sns
import matplotlib.pyplot as plt
sns.set(style='ticks', color_codes=True)
plt.figure(figsize=(14, 12))
sns.heatmap(df.astype(float).corr(), linewidths=0.1, square=True, linecolor='white', annot=True)
plt.show()
```

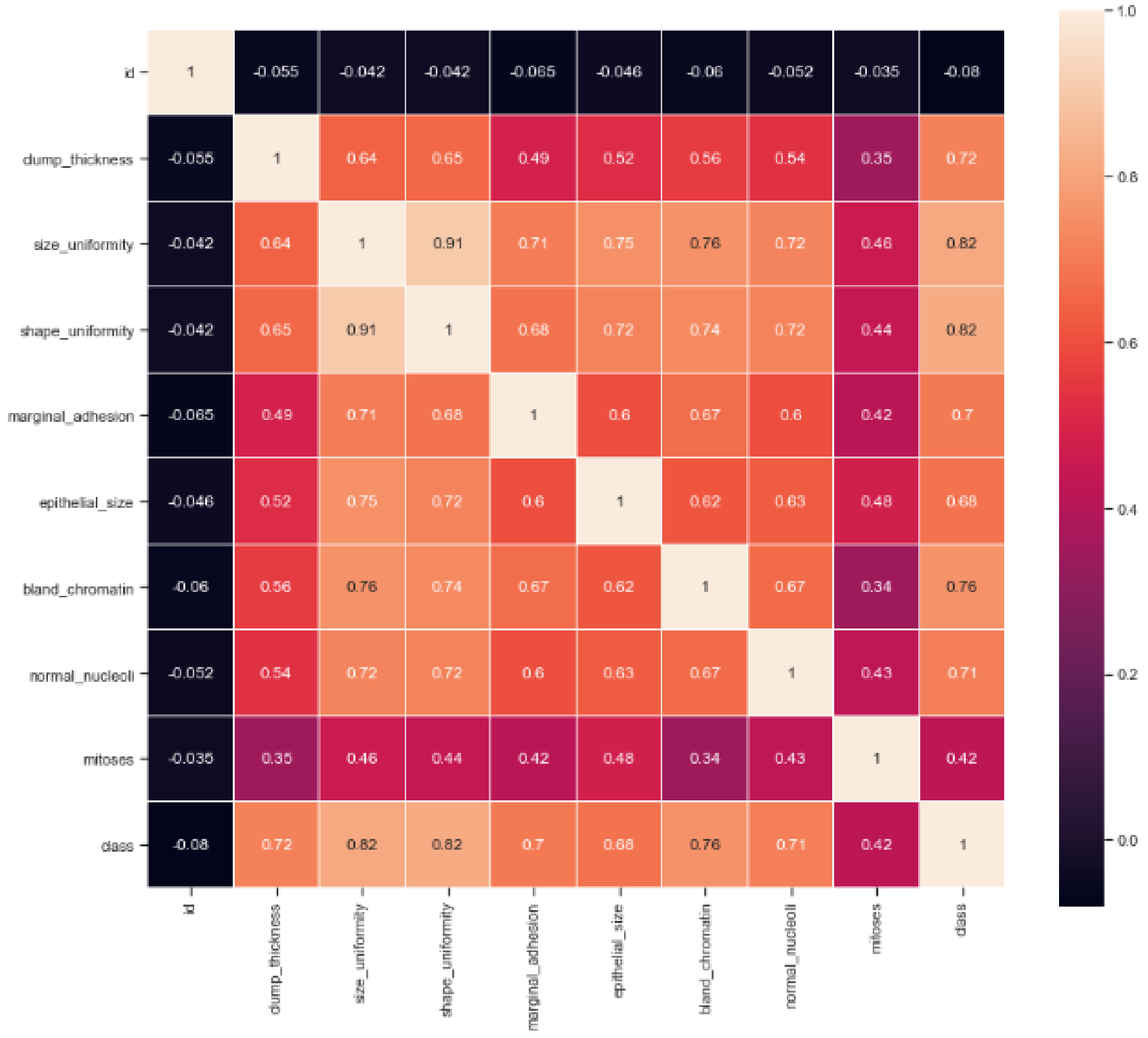
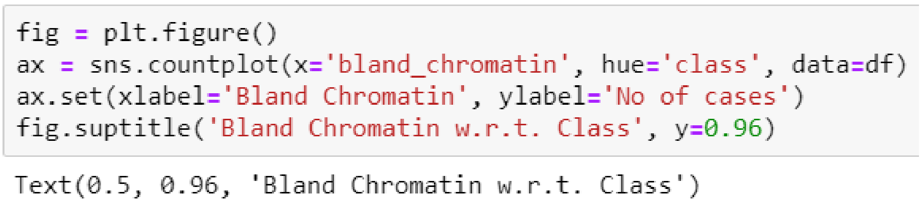


**Fig. 4** Heat Map

The Heat Map in Fig. 4 shows the correlation among all the attributes, where +1 shows the highest positive correlation and -1 being the negative correlation. Investigating the square where attribute `size_uniformity` of X-axis and `shape_uniformity` of Y -axis meet that is 0.91, which shows that these two attributes are highly co-related to each other, i.e., the value of `size_uniformity` increases when the value of `shape_uniformity` increases - had it been -0.91 again they are highly co-related but this time one would increase when the other would decreases.

Another way to explore the data would be to try and find correlation among the desease and the cause via bar-plots.

Selecting a column (`bland_chromatin`) on X axis and trying to predict the outputs on Y axis should produce a graph which shows how many people of each category in `bland_chromatin` will fall in class 2 or class 4 - class 2 means patient is in early stages of cancer, while class 4 is malevolent. Fig 5 shows the bar-chart.



```
fig = plt.figure()
ax = sns.countplot(x='bland_chromatin', hue='class', data=df)
ax.set(xlabel='Bland Chromatin', ylabel='No of cases')
fig.suptitle('Bland Chromatin w.r.t. Class', y=0.96)
```

```
Text(0.5, 0.96, 'Bland Chromatin w.r.t. Class')
```

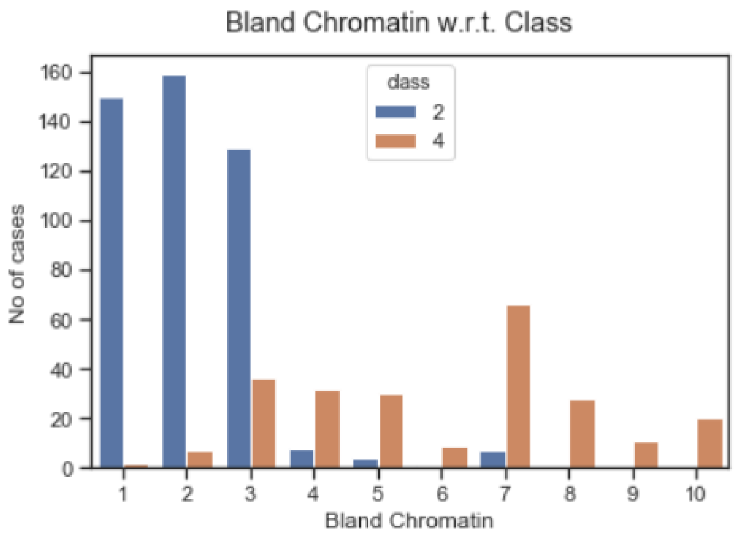
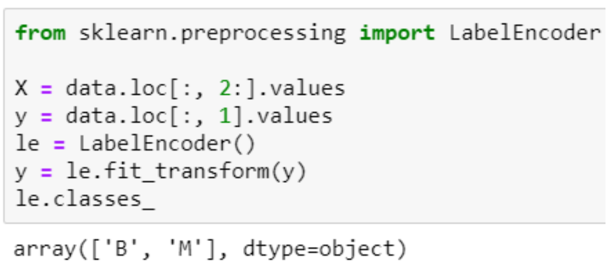


**Fig. 5** Bar-chart for correlation between `bland_chromatin` and class of cancer

The bar-chart depicts that when Bland Chromatin is in the range of 1 ,2 ,or 3, then 150, 160, 130 numbers, respectively, of patients are in benign stage; but as soon as the range increases from 3 to 7 , it appears that the number of patients fall in the danger area. Howver, few cases are still safe. Once the range exceeds 7, it was found that no patient was safe; hence between ranges 8 , 9 and 10 there were no case which was ruled the patients to be safe.

It was observed that the datset contains 569 rows and 32 columns. The 30 features of the datset are assigned to a NumPy array X - using a LabelEncoder object from sklearn libraries, the class labels are transformed from their original string representations of 'M' and 'B' into integers, shown in Fig 6:

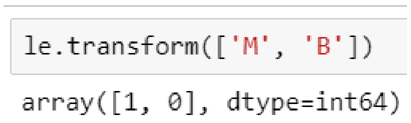


```
from sklearn.preprocessing import LabelEncoder

X = data.loc[:, 2:].values
y = data.loc[:, 1].values
le = LabelEncoder()
y = le.fit_transform(y)
le.classes_
```

```
array(['B', 'M'], dtype=object)
```

**Fig. 6** Benign and Malignant labels

The class labels (diagnosis) are encoded in an array y. Value of 1 indicates the cancer is malignant and 0 means benign. The malignant tumors are represented as class 1 and the benign tumors are represented as class 0, respectively. The mapping was verified by calling the transform method of the fitted LabelEncoder on two dummy class labels as shown in Fig. 7

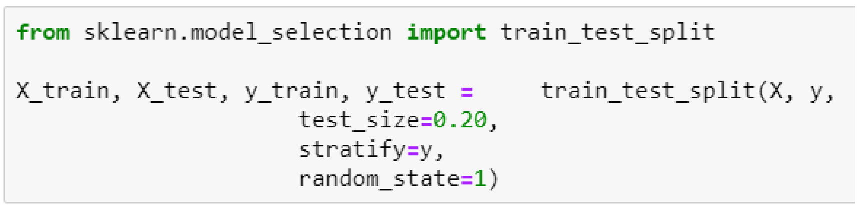


```
le.transform(['M', 'B'])
```

```
array([1, 0], dtype=int64)
```

**Fig. 7** Benign & Malignant Mapping

Next, the dataset was split into a training dataset with 80 percent of the data and a separate test dataset with 20 percent of the data (Fig. 8):



```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test =     train_test_split(X, y,
                     test_size=0.20,
                     stratify=y,
                     random_state=1)
```

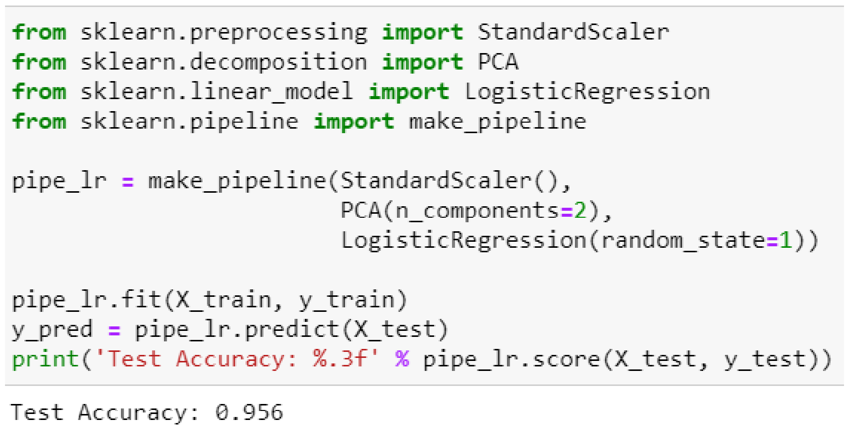
**Fig. 8** Split of train & test data

### 3.2.3 Feature Scaling

In most cases, datasets used will contain features which vary in magnitudes, units and range. However, since most of the machine learning algorithms use Eucledian distance between two data points in their computations, it is required to get all features to the same level of magnitude. This can be achieved by scaling. This means transforming the data so that it fits within a specific scale such as 0–100 or 0–1.

Learning algorithms require input features on the same scale for optimal performance. Hence, the columns in the Breast Cancer Wisconsin dataset need to be standardized before they can be fed into a linear classifier. An assumption is made to compress the data from the initial 30 dimensions onto a lower two-dimensional subspace via Principal Component Analysis (PCA), a feature extraction technique for dimensionality reduction.

To avoid separate steps for the fitting and transformation for the training and test datasets, the StandardScaler, PCA, and LogisticRegression objects are chained in a pipeline (Fig. 9):



```python
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import make_pipeline

pipe_lr = make_pipeline(StandardScaler(),
                        PCA(n_components=2),
                        LogisticRegression(random_state=1))

pipe_lr.fit(X_train, y_train)
y_pred = pipe_lr.predict(X_test)
print('Test Accuracy: %.3f' % pipe_lr.score(X_test, y_test))
```

```
Test Accuracy: 0.956
```

**Fig. 9** PCA, LogisticRegression and StandardScaler

In the above code snippet, the make_pipeline function takes an arbitrary number of scikit-learn transformers (objects that support the fit and transform methods as input), followed by a scikit-learn estimator that implements the fit and predict methods. In the preceding code there are two transformers, StandardScaler and PCA and a LogisticRegression estimator as inputs to the make_pipeline function, which constructs a scikit-learn Pipeline object from these objects.

The scikit-learn Pipeline can be thought of as a wrapper around those individual transformers and estimators. If the fit method of Pipeline is called, the data gets passed down a series of transformers via fit and transform calls on these intermediate steps until it arrives at the estimator object (the final element in a pipeline). The estimator then gets fitted to the transformed training data. The Pipeline object takes a list of tuples as input, where the first value in each tuple is an arbitrary identifier string that can be used to access the individual elements in the pipeline.

Once the fit method was executed, StandardScaler first performed fit and transform calls on the training data. Second, the transformed training data was passed on to the next object in the pipeline, PCA. Similar to the previous step, PCA also executed fit and transform on the scaled input data and passed it to the final element of the pipeline, the estimator. Finally, the LogisticRegression estimator was fit to the training data after it underwent transformations via StandardScaler and PCA. There is no limit to the number of intermediate steps in a pipeline; however, the last pipeline element has to be an estimator.

### 3.2.4 The holdout method

A classic and popular approach for estimating the generalization performance of machine learning models is holdout cross-validation. Using the holdout method, the initial dataset is split into a separate training and test dataset—the former is used for model training, and the latter is used to estimate its generalization performance. However, in typical machine learning applications, there is also interest in tuning and comparing different parameter settings to further improve the performance for making predictions on unseen data. This process is called model selection, where the term model selection refers to a given classification problem to select the optimal values of tuning parameters, also known as **hyperparameters**. However, if the same test dataset is re-used over and over again during model selection, it will become part of the training data and thus the model will be more likely to overfit.

An efficient way of using the holdout method for model selection is to separate the data into three parts: a training set, a validation set, and a test set. The training set is used to fit the different models, and the performance on the validation set is then used for the model selection. The advantage of having a test set that the model has not seen before during the training and model selection steps is that a less biased estimate can be obtained of its ability to generalize to new data.

A disadvantage of the holdout method is that the performance estimate may be very sensitive to how we partition the training set into the training and validation subsets; the estimate will vary for different samples of the data. A better method is k-fold cross-validation (CV).

### 3.2.5 k-fold cross validation to evaluate model performance

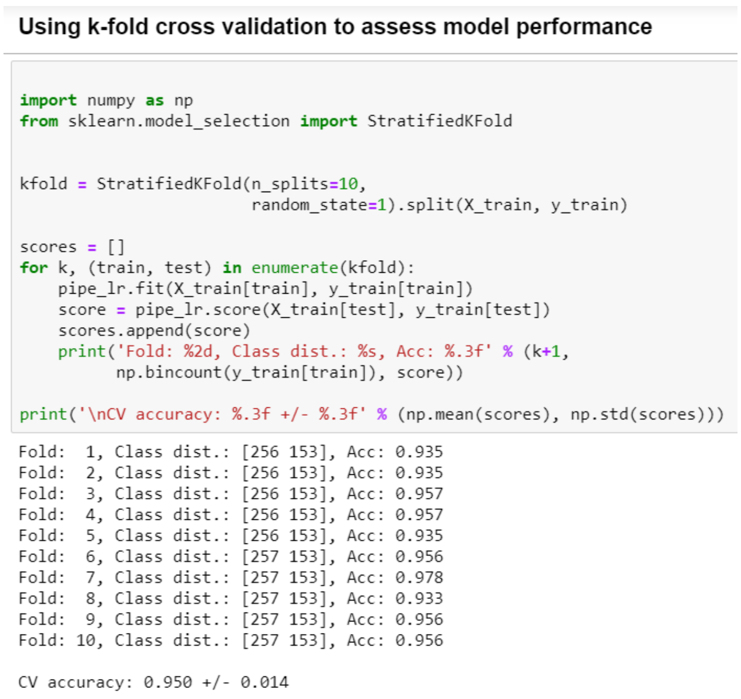
One of the key steps in building a machine learning model is to estimate its performance on data that the model hasn't seen before. To find an acceptable bias-variance trade-off, the model needs to be evaluated carefully.

In k-fold cross-validation, the training dataset is randomly split into k folds without replacement, where k — 1 folds are used for the model training, and one fold is used for performance evaluation. This procedure is repeated k times so that k models and performance estimates are obtained. Then the average performance is calculated of the models based on the different, independent folds to obtain a performance estimate that is less sensitive to the sub-partitioning of the training data compared to the holdout method. Typically, k-fold cross-validation is used for model tuning, that is, finding the optimal hyperparameter values that yields a satisfying generalization performance. Once satisfactory hyperparameter values have been found, the model can be retrained on the complete training set and obtain a final performance estimate using the independent test set. The rationale behind fitting a model to the whole training dataset after k-fold cross-validation is that providing more training samples to a learning algorithm usually results in a more accurate and robust model.

Since k-fold cross-validation is a resampling technique without replacement, the advantage of this approach is that each sample point will be used for training and validation (as part of a test fold) exactly once, which yields a lower-variance estimate of the model performance than the holdout method.

However, if working with relatively small training sets, it can be useful to increase the number of folds. If the value of k is increased then more training data will be used in each iteration, which results in a lower bias towards estimating the generalization performance by averaging the individual model estimates. However, large values of k will also increase the runtime of the cross-validation algorithm and yield estimates with higher variance, since the training folds will be more similar to each other. But when working with large datasets, a smaller value for k can be chosen, for example, k = 5, and still obtain an accurate estimate of the average performance of the model while reducing the computational cost of refitting and evaluating the model on the different folds.

A slight improvement over the standard k-fold cross-validation approach is stratified k-fold cross-validation, which can yield better bias and variance estimates, especially in cases of unequal class proportions, as has been shown in a study by Ron Kohavi [9]. The code for the implementation is shown in Fig. 10 and also included in the attached Jupyter Notebook and Appendix for this section.



Using k-fold cross validation to assess model performance

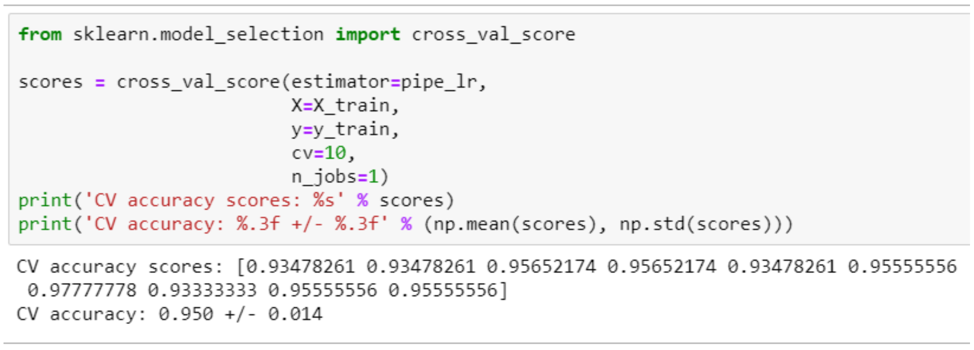
```
import numpy as np
from sklearn.model_selection import StratifiedKFold


kfold = StratifiedKFold(n_splits=10,
                        random_state=1).split(X_train, y_train)

scores = []
for k, (train, test) in enumerate(kfold):
    pipe_lr.fit(X_train[train], y_train[train])
    score = pipe_lr.score(X_train[test], y_train[test])
    scores.append(score)
    print('Fold: %2d, Class dist.: %s, Acc: %.3f' % (k+1,
          np.bincount(y_train[train]), score))

print('\nCV accuracy: %.3f +/- %.3f' % (np.mean(scores), np.std(scores)))
```

```
Fold:  1, Class dist.: [256 153], Acc: 0.935
Fold:  2, Class dist.: [256 153], Acc: 0.935
Fold:  3, Class dist.: [256 153], Acc: 0.957
Fold:  4, Class dist.: [256 153], Acc: 0.957
Fold:  5, Class dist.: [256 153], Acc: 0.935
Fold:  6, Class dist.: [257 153], Acc: 0.956
Fold:  7, Class dist.: [257 153], Acc: 0.978
Fold:  8, Class dist.: [257 153], Acc: 0.933
Fold:  9, Class dist.: [257 153], Acc: 0.956
Fold: 10, Class dist.: [257 153], Acc: 0.956

CV accuracy: 0.950 +/- 0.014
```

**Fig. 10** k-fold cross-validation

At first, the `StratifiedKfold` iterator from the `sklearn.model_selection` module with the `y_train` class labels in the training set, with a specified number of folds via the `n_splits` parameter. When the `kfold` iterator was used to loop through the k-folds, the returned indices helped to train to fit the logistic regression pipeline. Using the `pipe_lr` pipeline, it was ensured that the samples were scaled properly (for instance, standardized) in each iteration. Then the test indices were used to calculate the accuracy score of the model, which was collected in the scores list to calculate the average accuracy and the standard deviation of the estimate.

Although the previous code example was useful to illustrate how k-fold cross-validation works, scikit-learn also implements a k-fold cross-validation scorer, which allows for evaluation of the model using stratified k-fold cross-validation as shown in Fig 11:



```
from sklearn.model_selection import cross_val_score

scores = cross_val_score(estimator=pipe_lr,
                         X=X_train,
                         y=y_train,
                         cv=10,
                         n_jobs=1)
print('CV accuracy scores: %s' % scores)
print('CV accuracy: %.3f +/- %.3f' % (np.mean(scores), np.std(scores)))
```

```
CV accuracy scores: [0.93478261 0.93478261 0.95652174 0.95652174 0.93478261 0.95555556
 0.97777778 0.93333333 0.95555556 0.95555556]
CV accuracy: 0.950 +/- 0.014
```

**Fig. 11** k-fold cross-validation scorer

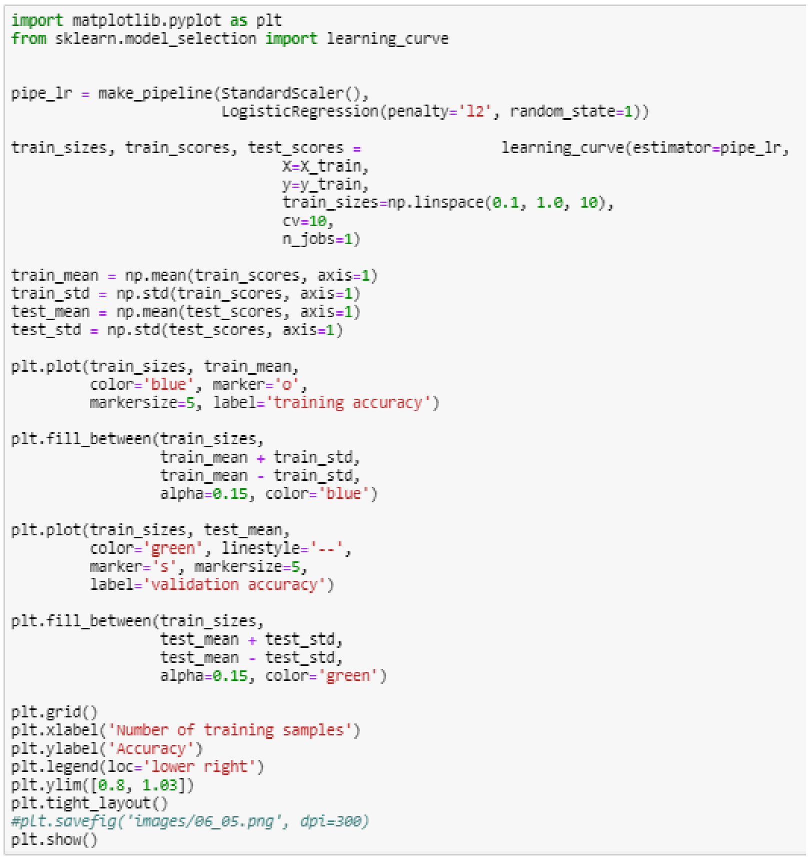
A detailed discussion of estimation of the variance of the generalization performance can be found in [10].

### 3.2.6 Debugging algorithms with learning and validation curves

This section will investigate effects of two very simple yet powerful diagnostic tools that help improve the performance of a learning algorithm: **learning curves** and **validation curves**.

If a model is too complex for a given training dataset—there are too many degrees of freedom or parameters in this model—the model tends to overfit the training data and does not generalize well to unseen data. Often, it can help to collect more training samples to reduce the degree of overfitting. However, in practice, it can often be very expensive or simply not feasible to collect more data. By plotting the model training and validation accuracies as functions of the training set size, it can be easily detected whether the model suffers from high variance or high bias, and whether the collection of more data could help address this problem.

Following code in Fig. 12a used the learning curve function of scikit-learn to evaluate the model, which in turn gave the plot of Fig. 12b as output.



```
import matplotlib.pyplot as plt
from sklearn.model_selection import learning_curve


pipe_lr = make_pipeline(StandardScaler(),
                        LogisticRegression(penalty='l2', random_state=1))

train_sizes, train_scores, test_scores =                learning_curve(estimator=pipe_lr,
                               X=X_train,
                               y=y_train,
                               train_sizes=np.linspace(0.1, 1.0, 10),
                               cv=10,
                               n_jobs=1)

train_mean = np.mean(train_scores, axis=1)
train_std = np.std(train_scores, axis=1)
test_mean = np.mean(test_scores, axis=1)
test_std = np.std(test_scores, axis=1)

plt.plot(train_sizes, train_mean,
         color='blue', marker='o',
         markersize=5, label='training accuracy')

plt.fill_between(train_sizes,
                 train_mean + train_std,
                 train_mean - train_std,
                 alpha=0.15, color='blue')

plt.plot(train_sizes, test_mean,
         color='green', linestyle='--',
         marker='s', markersize=5,
         label='validation accuracy')

plt.fill_between(train_sizes,
                 test_mean + test_std,
                 test_mean - test_std,
                 alpha=0.15, color='green')

plt.grid()
plt.xlabel('Number of training samples')
plt.ylabel('Accuracy')
plt.legend(loc='lower right')
plt.ylim([0.8, 1.03])
plt.tight_layout()
#plt.savefig('images/06_05.png', dpi=300)
plt.show()
```

**Fig. 12a** Diagnosing bias and variance problems with learning curves

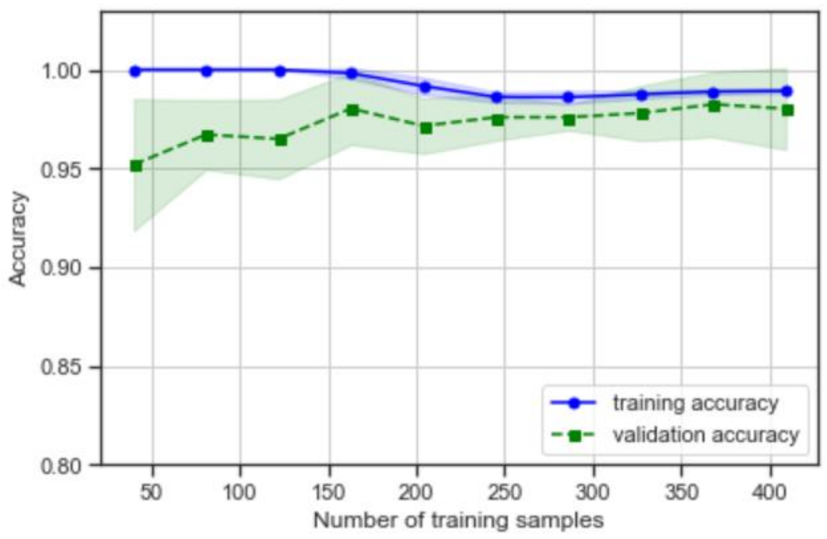


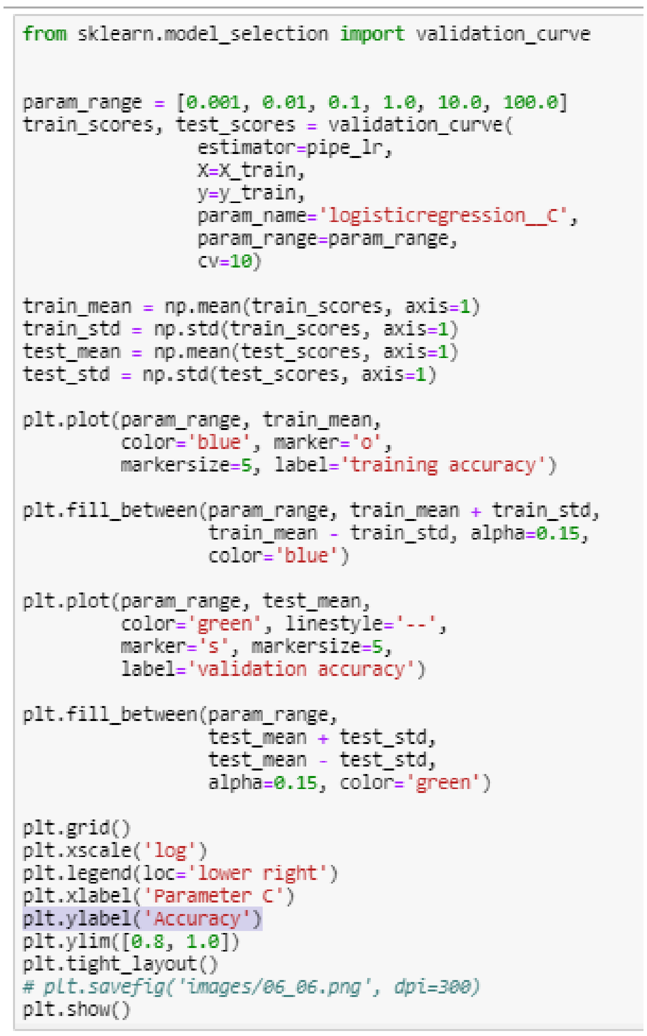
**Fig. 12b** Learning curve plot

The `train_sizes` parameter in the `learning_curve` function control the absolute or relative number of training samples that are used to generate the learning curves. The value of `train_sizes=np.linspace(0.1, 1.0, 10)` was set to use 10 evenly spaced, relative intervals for the training set sizes. By default, the `learning_curve` function uses stratified k-fold cross-validation to calculate the cross-validation accuracy of a classifier, and k=10 was assigned via the `cv` parameter for 10-fold stratified cross-validation. Then, the average accuracies were calculated from the returned cross-validated training and test scores for the different sizes of the training set, which were plotted using Matplotlib's plot function. The standard deviation of the average accuracy was added to the plot using the `fill_between` function to indicate the variance of the estimate.

In the preceding learning curve plot (Fig. 12b), the model performs quite well on both the training and validation dataset if it had seen more than 250 samples during training. It is also apparent that the training accuracy increases for training sets with fewer than 250 samples, and the gap between validation and training accuracy widens—an indicator of an increasing degree of **overfitting**. The relatively small, but visible gap between training and cross-validation curves is an indicator of slight overfitting.

### 3.2.7 Overfitting and Underfitting: Validation Curves

Validation curves are a useful tool for improving the performance of a model by addressing issues such as overfitting or underfitting. Validation curves are related to learning curves, but instead of plotting the training and test accuracies as functions of the sample size, the values of the model parameters are varied. The code snippet is shown in Fig 13a and the validation curves in Fig. 13b. In this code, the inverse regularization parameter C in logistic regression is tweaked.



```python
from sklearn.model_selection import validation_curve


param_range = [0.001, 0.01, 0.1, 1.0, 10.0, 100.0]
train_scores, test_scores = validation_curve(
                estimator=pipe_lr,
                X=X_train,
                y=y_train,
                param_name='logisticregression__C',
                param_range=param_range,
                cv=10)

train_mean = np.mean(train_scores, axis=1)
train_std = np.std(train_scores, axis=1)
test_mean = np.mean(test_scores, axis=1)
test_std = np.std(test_scores, axis=1)

plt.plot(param_range, train_mean,
         color='blue', marker='o',
         markersize=5, label='training accuracy')

plt.fill_between(param_range, train_mean + train_std,
                 train_mean - train_std, alpha=0.15,
                 color='blue')

plt.plot(param_range, test_mean,
         color='green', linestyle='--',
         marker='s', markersize=5,
         label='validation accuracy')

plt.fill_between(param_range,
                 test_mean + test_std,
                 test_mean - test_std,
                 alpha=0.15, color='green')

plt.grid()
plt.xscale('log')
plt.legend(loc='lower right')
plt.xlabel('Parameter C')
plt.ylabel('Accuracy')
plt.ylim([0.8, 1.0])
plt.tight_layout()
# plt.savefig('images/06_06.png', dpi=300)
plt.show()
```

**Fig. 13a** Generating validation curves

The code in Fig. 13a generated the validation curve plot for the parameter C (Fig. 13b)

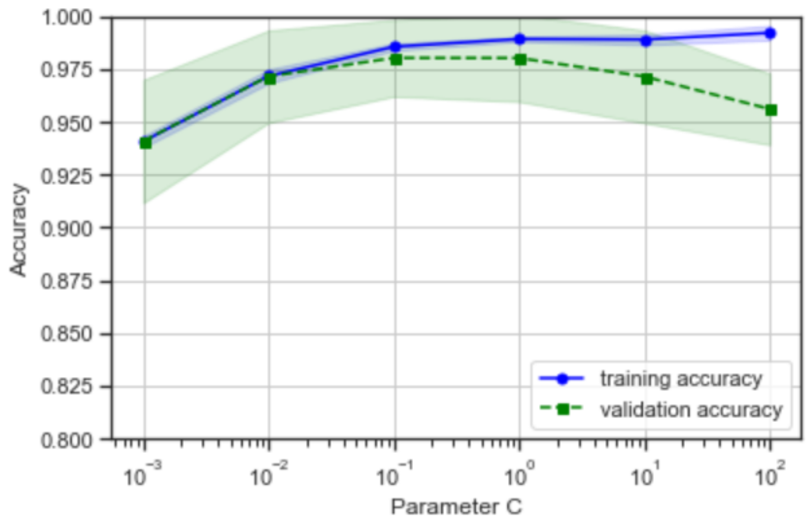


**Fig. 13b** Validation curves

The `validation_curve` function uses stratified k-fold cross-validation by default to estimate the performance of the classifier. Inside the `validation_curve` function, the parameter to be evaluated is specified. In this case, it is C, the inverse regularization parameter of the `LogisticRegression` classifier, which was written as `'logisticregression__C'` to access the `LogisticRegression` object inside the scikit-learn pipeline for a specified value range that was set via the `param_range` parameter. Similar to the learning curve example in the previous section, the average training and cross-validation accuracies were plotted and so were the corresponding standard deviations.

Although the differences in the accuracy for varying values of C are somewhat subtle, the model slightly underfits the data when the regularization strength, i.e., small values of C is increased. However, for large values of C, it means lowering the strength of regularization, so the model tends to slightly overfit the data. In this case, the sweet spot appears to be between 0.01 and 0.1 of the C value.

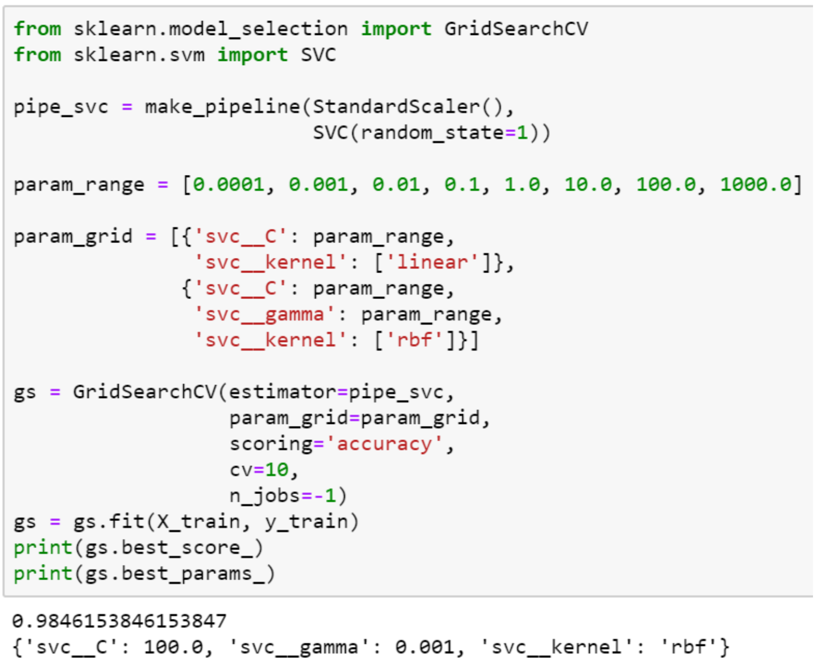
### 3.2.8 Tuning ML Model via Grid Search

In machine learning, there are two types of parameters: those that are learned from the training data, for example, the weights in logistic regression, and the parameters of a learning algorithm that are optimized separately. The latter are the tuning parameters, also called hyperparameters, of a model, for example, the regularization parameter in logistic regression or the depth parameter of a decision tree.

This section focusses on hyperparameter optimization technique called grid search that can further help improve the performance of a model by finding the optimal combination of hyperparameter values.

#### *3.2.8.1 Tuning Hyperparameters via Grid Search*

Grid search is a brute-force exhaustive search paradigm where a list of values are specified for different hyperparameters, and the computer evaluates the model performance for each combination of those to obtain the optimal combination of values from this set. Following Fig. 14 shows the code snippet of the grid search algorithm for this model.



```python
from sklearn.model_selection import GridSearchCV
from sklearn.svm import SVC

pipe_svc = make_pipeline(StandardScaler(),
                         SVC(random_state=1))

param_range = [0.0001, 0.001, 0.01, 0.1, 1.0, 10.0, 100.0, 1000.0]

param_grid = [{'svc__C': param_range,
               'svc__kernel': ['linear']},
              {'svc__C': param_range,
               'svc__gamma': param_range,
               'svc__kernel': ['rbf']}]

gs = GridSearchCV(estimator=pipe_svc,
                  param_grid=param_grid,
                  scoring='accuracy',
                  cv=10,
                  n_jobs=-1)
gs = gs.fit(X_train, y_train)
print(gs.best_score_)
print(gs.best_params_)
```

```
0.9846153846153847
{'svc__C': 100.0, 'svc__gamma': 0.001, 'svc__kernel': 'rbf'}
```

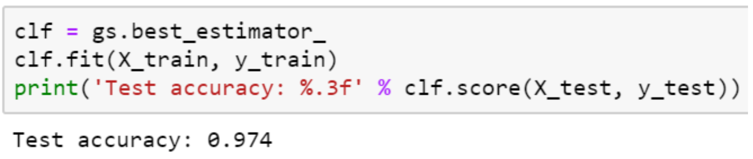
**Fig. 14** Grid search code

In Fig. 14 a `GridSearchCV` object was initiated from the `sklearn.model_selection` module to train and tune a **Support Vector Machine** (**SVM**) pipeline. The `param_grid` parameter was set of `GridSearchCV` to a list of dictionaries to specify the parameters to be tuned. For the linear SVM, only the inverse regularization parameter C was evaluated; for the RBF kernel SVM, both the `svc__C` and `svc__gamma` parameters were tuned. The `svc__gamma` parameter is specific to kernel SVMs.

After the training data was used to perform the grid search, the score of the best-performing model via the `best_score_ attribute` was obtained. The parameters that can be accessed via the `best_params_` attribute were also investigated. In this particular case, the RBF-kernel SVM model with `svc__C = 100.0` yielded the best k-fold cross-validation accuracy of ~98.5 %.

> **Note**: Although grid search is a powerful approach for finding the optimal set of parameters, the evaluation of all possible parameter combinations is also computationally *very expensive*. An alternative approach to sampling different parameter combinations using scikit-learn is randomized search using the `RandomizedSearchCV` class in scikit-learn.

Finally, an estimate was done for the performance of the best-selected model, which is available via the `best_estimator_` attribute of the `GridSearchCV` object (Fig 15) to obtain a test accuracy of 97.4%.



```
clf = gs.best_estimator_
clf.fit(X_train, y_train)
print('Test accuracy: %.3f' % clf.score(X_test, y_test))
```

```
Test accuracy: 0.974
```

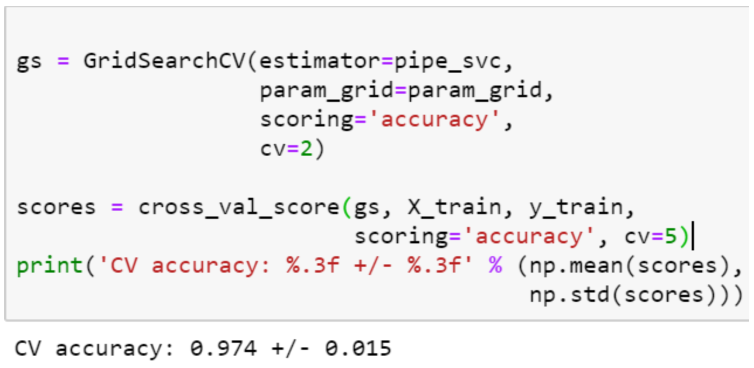
**Fig. 15** Model Performance

### *3.2.8.2 Algorithm Selection: Nested Cross-validation*

k-fold cross-validation in combination with grid search is a useful approach for fine-tuning the performance of a machine learning model by varying its hyperparameter values. However, another popular machine learning algorithm choice process is nested cross-validation. A study on the bias in error estimation by Varma and Simon [11] concluded that the true error of the estimate is almost unbiased relative to the test set when nested cross-validation is used.

In nested cross-validation, there is an outer k-fold cross-validation loop to split the data into training and test folds, and an inner loop is used to select the model using k-fold cross-validation on the training fold. After model selection, the test fold is then used to evaluate the model performance.

Nested cross-validation was performed in scikit-learn as follows (Fig. 16):

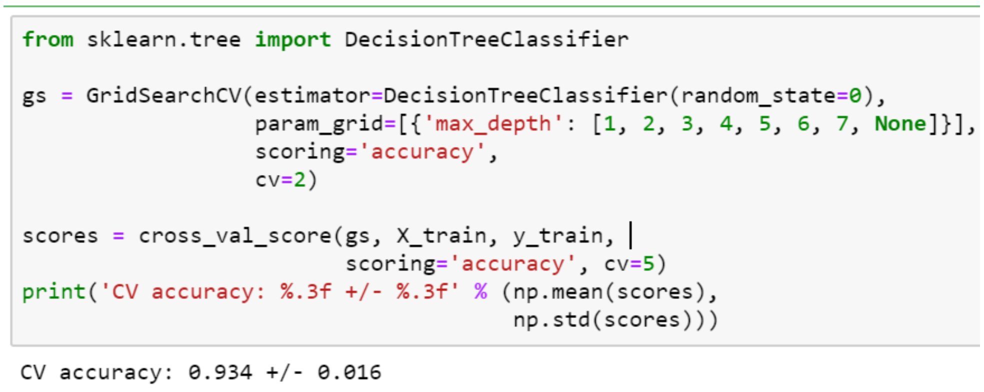


```
gs = GridSearchCV(estimator=pipe_svc,
                  param_grid=param_grid,
                  scoring='accuracy',
                  cv=2)

scores = cross_val_score(gs, X_train, y_train,
                         scoring='accuracy', cv=5)
print('CV accuracy: %.3f +/- %.3f' % (np.mean(scores),
                                      np.std(scores)))
```

```
CV accuracy: 0.974 +/- 0.015
```

**Fig. 16** Nested cross-validation

The returned average cross-validation accuracy gives a good estimate of what to expect if the hyperparameters of a model are tuned and used on unseen data. For example, the nested cross-validation approach can be used to compare an SVM model to a simple decision tree classifier; for simplicity, only its depth parameter was tuned (Fig. 17):



```
from sklearn.tree import DecisionTreeClassifier

gs = GridSearchCV(estimator=DecisionTreeClassifier(random_state=0),
                  param_grid=[{'max_depth': [1, 2, 3, 4, 5, 6, 7, None]}],
                  scoring='accuracy',
                  cv=2)

scores = cross_val_score(gs, X_train, y_train,
                         scoring='accuracy', cv=5)
print('CV accuracy: %.3f +/- %.3f' % (np.mean(scores),
                                      np.std(scores)))
```

```
CV accuracy: 0.934 +/- 0.016
```

**Fig. 17** Nested cross-validation performance

**Conclusion**: From Fig 16 and 17, it is inferred that the nested cross-validation performance of the SVM model at 97.4 % is notably better than the performance of the decision tree at 93.4 percent and hence, it is expected that it might be the better choice to classify new data that comes from the same population as this particular dataset.

### 3.2.9 Confusion Matrix

Confusion Matrix is a matrix that states the performance of a learning algorithm. The confusion matrix is simply a square matrix that reports the counts of the T**rue positive (TP), True negative (TN), False positive (FP)**, and **False negative (FN)** predictions of a classifier as shown in the Fig 18.

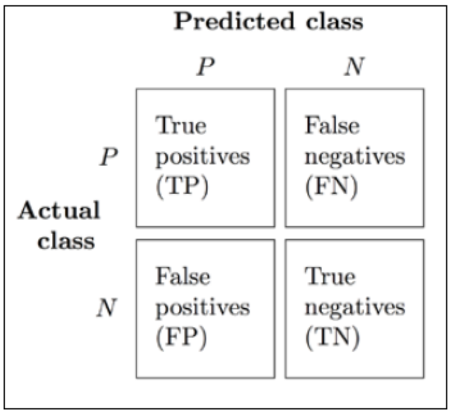
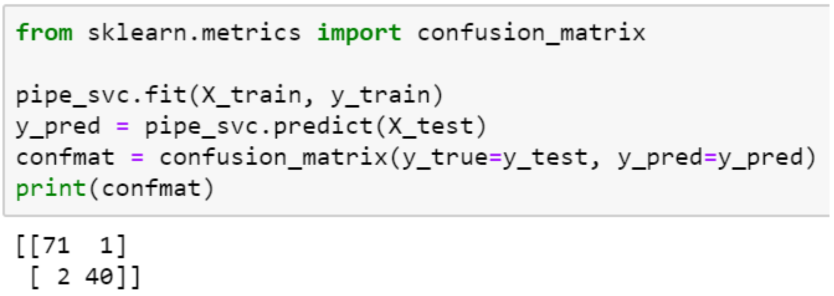


**Fig. 18** Confusion Matrix

Although these metrics can be computed manually by comparing the true and predicted class labels, scikit-learn provides a convenient `confusion_matrix` function that can be used, as follows (Fig. 19a):

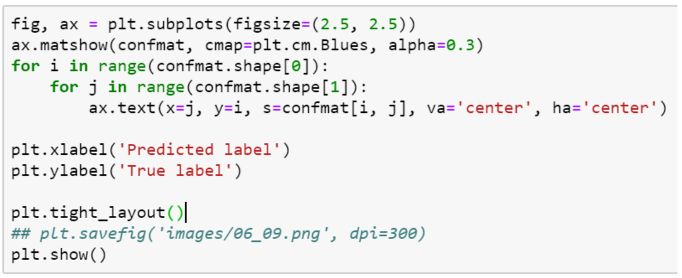


```
from sklearn.metrics import confusion_matrix

pipe_svc.fit(X_train, y_train)
y_pred = pipe_svc.predict(X_test)
confmat = confusion_matrix(y_true=y_test, y_pred=y_pred)
print(confmat)
```

```
[[71  1]
 [ 2 40]]
```

**Fig. 19a** Array depicting errors by the classifier on the test dataset

The returned array (in Fig. 19a) after executing the code provides information about the different types of error the classifier made on the test dataset. This information can be mappped onto the confusion matrix illustration in Fig 18 using Matplotlib's `matshow` function (Fig 19b):



```
fig, ax = plt.subplots(figsize=(2.5, 2.5))
ax.matshow(confmat, cmap=plt.cm.Blues, alpha=0.3)
for i in range(confmat.shape[0]):
    for j in range(confmat.shape[1]):
        ax.text(x=j, y=i, s=confmat[i, j], va='center', ha='center')

plt.xlabel('Predicted label')
plt.ylabel('True label')

plt.tight_layout()
## plt.savefig('images/06_09.png', dpi=300)
plt.show()
```

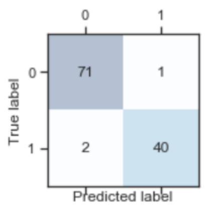
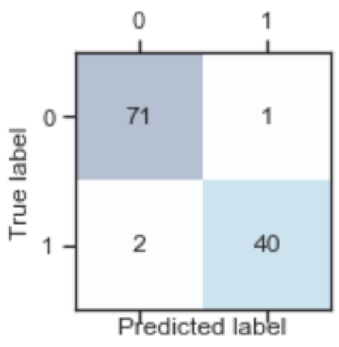


**Fig. 19b** Generating the Confusion Matrix with Matplotlib

The generated confusion matrix in Fig. 19b makes it easier to interpret the information.



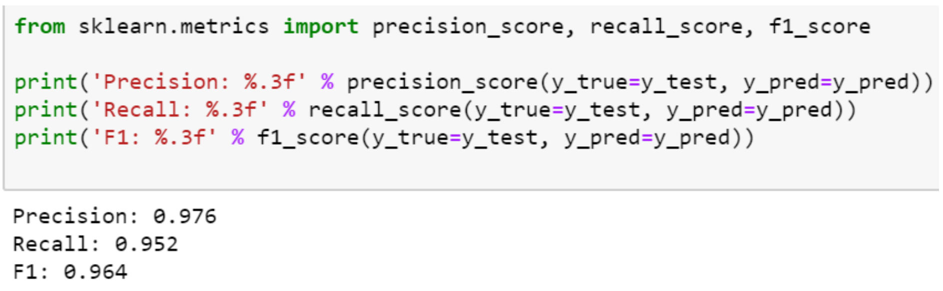
Assuming that class 1 (malignant) is the positive class in this example, the model correctly classified 71 of the samples that belong to class 0 (TNs) and 40 samples that belong to class 1 (TPs), respectively. However, the model also incorrectly misclassified two samples from class 1 as class 0 (FN), and it predicted that one sample is malignant although it is a benign tumor (FP).

### 3.2.10 Performance Metrics

The previous section generated information to calculate various error metrics. The prediction error (**ERR**) and accuracy (**ACC**) provide general information about how many samples are misclassified. The error can be understood as the sum of all false predictions divided by the number of total predications, and the accuracy is calculated as the sum of correct predictions divided by the total number of predictions, respectively. The True positive rate (**TPR**) and False positive rate (**FPR**) are performance metrics that are especially useful for imbalanced class problems.

In tumor diagnosis there are more concerns about the detection of malignant tumors in order to help a patient with the appropriate treatment. However, it is also important to decrease the number of benign tumors that were incorrectly classified as malignant (FPs) to not unnecessarily concern a patient. In contrast to the FPR, the TPR provides useful information about the fraction of positive (or relevant) samples that were correctly identified out of the total pool of positives (P). The performance metrics precision (**PRE**) and recall (**REC**) are related to those true positive and negative rates.

These scoring metrics are all implemented in scikit-learn and can be imported from the `sklearn.metrics` module as shown in the following snippet (Fig. 20):

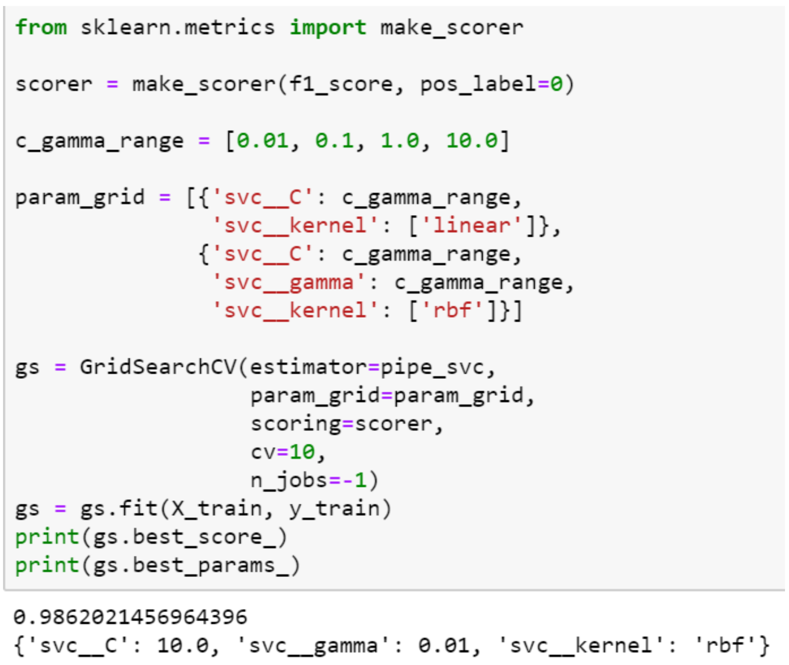


```
from sklearn.metrics import precision_score, recall_score, f1_score

print('Precision: %.3f' % precision_score(y_true=y_test, y_pred=y_pred))
print('Recall: %.3f' % recall_score(y_true=y_test, y_pred=y_pred))
print('F1: %.3f' % f1_score(y_true=y_test, y_pred=y_pred))
```

```
Precision: 0.976
Recall: 0.952
F1: 0.964
```

**Fig. 20** Scoring Metrics

The positive class in scikit-learn is the class that is labelled as class 1. In order to specify a different positive label, a scorer can be constructed via the `make_scorer` function, which can then be directly provided as an argument to the scoring parameter in `GridSearchCV` (in this example, using the `f1_score` as a metric) as shown in Fig 21:



```
from sklearn.metrics import make_scorer

scorer = make_scorer(f1_score, pos_label=0)

c_gamma_range = [0.01, 0.1, 1.0, 10.0]

param_grid = [{'svc__C': c_gamma_range,
               'svc__kernel': ['linear']},
              {'svc__C': c_gamma_range,
               'svc__gamma': c_gamma_range,
               'svc__kernel': ['rbf']}]

gs = GridSearchCV(estimator=pipe_svc,
                  param_grid=param_grid,
                  scoring=scorer,
                  cv=10,
                  n_jobs=-1)
gs = gs.fit(X_train, y_train)
print(gs.best_score_)
print(gs.best_params_)
```

```
0.9862021456964396
{'svc__C': 10.0, 'svc__gamma': 0.01, 'svc__kernel': 'rbf'}
```

**Fig. 21** Construction of a scorer with scikit-learn

### 3.2.11 Receiver Operating Characteristic (ROC)

Receiver Operating Characteristic (ROC) graphs are useful tools to select models for classification based on their performance with respect to the FPR and TPR, which are computed by shifting the decision threshold of the classifier. The diagonal of an ROC graph can be interpreted as random guessing, and classification models that fall below the diagonal are considered as worse than random guessing. A perfect classifier would fall into the top left corner of the graph with a TPR of 1 and an FPR of 0. Based on the ROC curve, the ROC Area Under the Curve (ROC AUC) can be computed to characterize the performance of a classification model. Similar to ROC curves, the precision-recall curves for different probability thresholds can be computed of a classifier. A function for plotting those precision-recall curves is also implemented in scikit-learn and is documented at http://scikitlearn.org/stable/modules/generated/sklearn.metrics.precision_recall_curve.html .

Executing the following code example, an ROC curve of a classifier was plotted that only uses two features from the Breast Cancer Wisconsin dataset to predict whether a tumor is benign or malignant. This makes the classification task more challenging for the classifier so that the resulting ROC curve becomes visually more interesting. For reasons of simplicity the number of folds in the `StratifiedKFold` validator has also been reduced to three. The ROC curve is as follows (Fig. 22) and the code is attached in ipynb files.

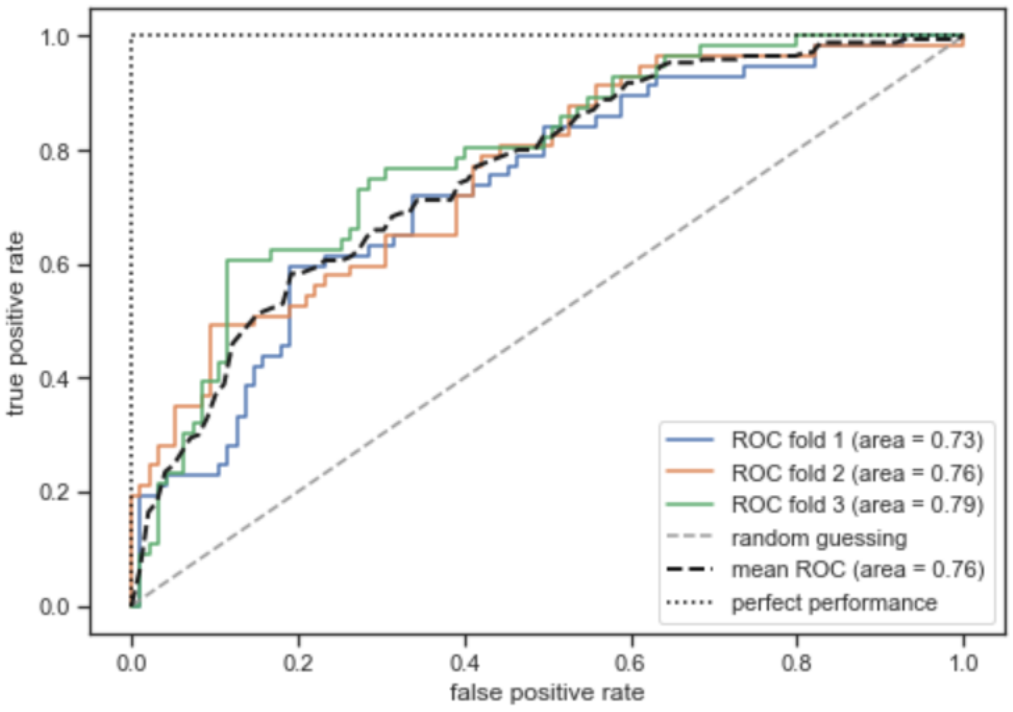


**Fig. 22** ROC curves for the Wisconsin Breast Cancer dataset

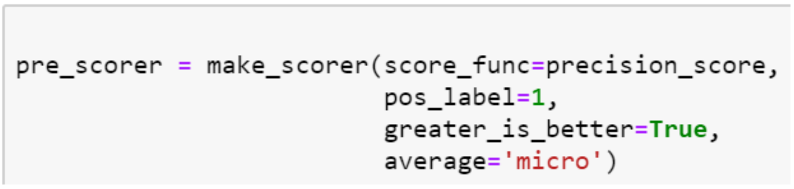
The ROC curve in Fig 22 indicates that there is a certain degree of variance between the different folds, and the average ROC AUC (0.76) falls between a perfect score (1.0) and random guessing (0.5).

### 3.2.12 Scoring metrics for multiclass classification

Scikit-learn also implements macro and micro averaging methods to extend the previously discussed scoring metrics to multiclass problems via **One-versus-All (OvA)** classification. The micro-average is calculated from the individual TPs, TNs, FPs, and FNs of the system.

Micro-averaging is useful for weighing each instance or prediction equally, whereas macro-averaging weights all classes equally to evaluate the overall performance of a classifier with regard to the most frequent class labels.  If binary performance metrics are used to evaluate multiclass classification models in scikit-learn, a normalized or weighted variant of the macro-average is used by default. The weighted macro-average is calculated by weighting the score of each class label by the number of true instances when calculating the average.

The weighted macro-average is useful if class imbalances are being dealt with, that is, different numbers of instances for each label. While the weighted macro-average is the default for multiclass problems in scikitlearn, it is possible to specify the averaging method via the average parameter inside the different scoring functions that can be imported from the `sklearn.metrics` module, for example, the `precision_score` or `make_scorer` functions (Fig 23):



```
pre_scorer = make_scorer(score_func=precision_score,
                         pos_label=1,
                         greater_is_better=True,
                         average='micro')
```

**Fig. 23** Scoring Metrics for Multiclass Classification

### 3.2.12.1 *Class Imbalance*

Class imbalance is a quite common problem when working with real-world data—samples from one class or multiple classes are over-represented in a dataset. If the breast cancer dataset consisted of 90 percent healthy patients, accuracy of about 90% on the test dataset can be achieved by just predicting the majority class (benign tumor) for all samples, without the help of a supervised machine learning algorithm. Thus, training a model on such a dataset that achieves approximately 90% test accuracy would mean the model did not learn anything useful from the features provided in this dataset.

An imbalanced dataset was created from the breast cancer dataset, which originally consisted of 357 benign tumors (class 0) and 212 malignant tumors (class 1) as in Fig 24:

```
X_imb = np.vstack((X[y == 0], X[y == 1][:40]))
y_imb = np.hstack((y[y == 0], y[y == 1][:40]))
```

**Fig. 24** Imbalanced Dataset creation

In Fig 24, all 357 benign tumor samples were taken and stacked with the first 40 malignant samples to create a stark class imbalance. If computation of the accuracy of a model that always predicts the majority class (benign,class 0) was to be done, then that would achieve a prediction accuracy of approximately 90 percent (Fig. 25):

```
y_pred = np.zeros(y_imb.shape[0])
np.mean(y_pred == y_imb) * 100
```

```
89.92443324937027
```

**Fig. 25** Accuracy of a model that always predicts the majority class

When classifiers are fitted on such datasets, it would make sense to focus on other metrics than accuracy when comparing different models, such as precision, recall, the ROC curve etc. in our application. For instance, the priority might be to identify the majority of patients with malignant cancer patients to recommend an additional screening, then *recall* should be the metric of choice.

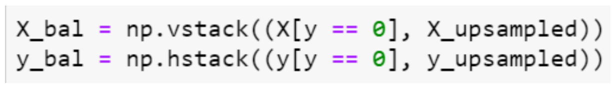
The scikit-learn library implements a simple `resample` function that can help with the up-sampling of the minority class by drawing new samples from the dataset with replacement. The following code takes the minority class from the imbalanced breast cancer dataset (class 1) and repeatedly draw new samples from it until it contains the same number of samples as class label 0 (Fig 26a):

```
from sklearn.utils import resample

print('Number of class 1 samples before:', X_imb[y_imb == 1].shape[0])

X_upsampled, y_upsampled = resample(X_imb[y_imb == 1],
                                    y_imb[y_imb == 1],
                                    replace=True,
                                    n_samples=X_imb[y_imb == 0].shape[0],
                                    random_state=123)

print('Number of class 1 samples after:', X_upsampled.shape[0])
```

```
Number of class 1 samples before: 40
Number of class 1 samples after: 357
```

**Fig. 26a** Resampling of minority class

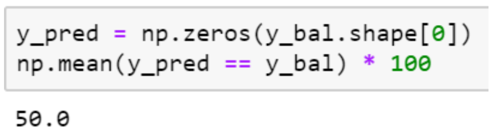
After resampling, the original class 0 samples were stacked with the upsampled class 1 subset to obtain a balanced dataset as follows (Fig. 26b)



```
X_bal = np.vstack((X[y == 0], X_upsampled))
y_bal = np.hstack((y[y == 0], y_upsampled))
```

**Fig. 26b** Stacked Upsampled Class

A majority vote prediction rule would only achieve 50 percent accuracy (Fig 27):



```
y_pred = np.zeros(y_bal.shape[0])
np.mean(y_pred == y_bal) * 100
```

```
50.0
```

**Fig. 27** Prediction

## 3.3 Conclusion:

This section explored deep learning methods included chaining different transformation techniques and classifiers in convenient model pipelines that helped train and evaluate machine learning models more efficiently. Those pipelines were used to perform k-fold cross-validation, one of the essential techniques for model selection and evaluation. Using k-fold cross-validation, learning and validation curves were used to diagnose the common problems of learning algorithms such as overfitting and underfitting. Using grid search, the model was further fine-tuned. Also were addressed onfusion matrix and various performance metrics that can be useful to further optimize a model's performance for a specific problem task.

## 3.4 Forward looking Comments: Quantum Machine Learning

Machine Learning and Artificial Intelligence has taken the industry by storm. Since its industry wide adaptation started to accelerate in 2014, there has been an 80% rise in adaptation of Ml/AI in industry for various functionalities and characteristics of datasets. Because of the promise and power offered, the adaptation of AI/ML is likely going to continue to be highly customized, different data characteristics requiring different solutions and methods.

As an exercise the PCA of a reduced WBCD was tried on a Quantum Machine Learning Algorithm running on classical machine (the author's laptop) using IBM Q's Qiskit Python library (Fig. 28) – the code is attached in Appendix and also supplied in the file "QML_WBCD.ipynb":

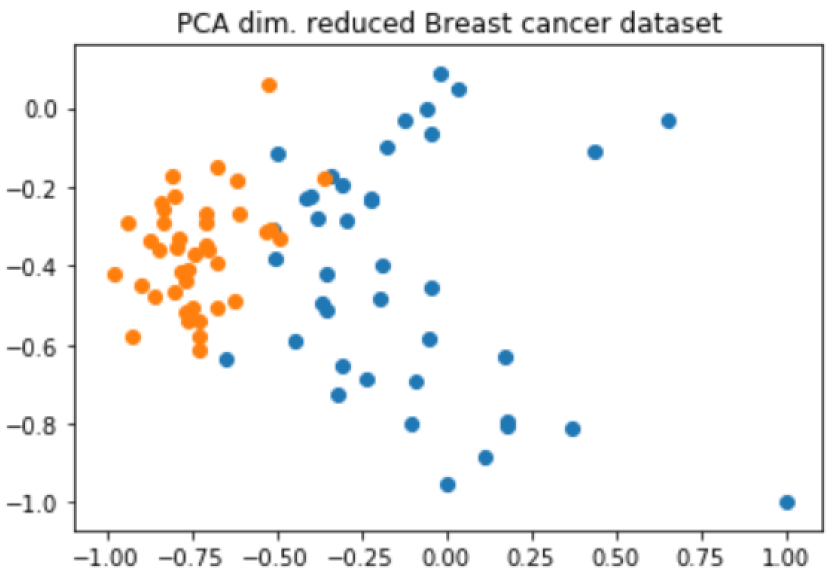


**Fig. 27** PCA of reduced breast cancer dataset

The results show that so far quantum computers are able to simulate similar results to a classical computer. The data classification is shown and the different colors represent whether a datapoint classifies as cancerous or not.

## References

[1] Cisco Data Analytics Strategy: https://data-analytics.cisco.com/#/
[2] Ian Goodfellow et. Al., Generative Adversarial Networks. arXiv. June 10 , 2014: https://arxiv.org/abs/1406.2661
[3] Official Keras documentation, https://keras.io/layers/core/
[4] Sandro Skansi, Introduction to Deep Learning. Springer, 2018
[5] S. Raschka and V. Mirjalili, Python Machine Learning, 2nd Edition. Packt, 2017
[6] Ian Goodfellow et. Al., Deep Learning. MIT Press, 2014
[7] MAPE: https://stats.stackexchange.com/questions/58391/meanabsolute-percentage-error-mapein-scikit-learn and RMSE: https://stackoverflow.com/questions/16774849/mean-squared-error-innumpy
[8] Aurelién Géron, Hands-on Machine Learning with Scikit-Learn and TensorFlow, O'Reilly, March, 2017
[9] Ron Kohavi, A Study of Cross-Validation and Bootstrap for Accuracy Estimation and Model Selection, International Joint Conference on Artificial Intelligence (IJCAI), 14 (12): 1137-43, 1995
[10] M. Markatou, H. Tian, S. Biswas and G. M. Hripcsak, Analysis of Variance of Cross-validation Estimators of the Generalization Error, Journal of Machine Learning Research, 6: 1127-1168, 2005
[11] S. Varma and R. Simon, Bias in Error Estimation When Using Cross-validation for Model Selection, BMC Bioinformatics, 7(1): 91, 2006)
[12] IBM Q's Qiskit: https://github.com/Qiskit/

# Appendix 1

**Magic Commands**

Magic commands (those that start with `%` ) are commands that modify a configuration of Jupyter Notebooks. A number of magic commands are available by default (see list here (http://ipython.readthedocs.io/en/stable/interactive/magics.html))--and many more can be added with extensions. The magic command added in this section allows `matplotlib` to display our plots directly on the browser instead of having to save them on a local file.

In [43]:

```
%matplotlib inline
```

# Section 2.2.1: Exploring Bitcoin Dataset

Exploration of the Bitcoin dataset. Step 1 is to import the required libraries.

## Introduction

In [2]:

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

Import the custom set of normalization functions via normalization.py file.

In [10]:

```
import normalizations
```

Then the dataset is loaded as a pandas `DataFrame` to make it easier to compute basic properties from the dataset and to clean any irregularities.

In [11]:

```
bitcoin = pd.read_csv('C:\Knowledgebase\Certs\DataScience\BitCoin\data\cryptocurrency_price
```

the dataset contains 8 variables (i.e. columns) as follows:

- `date` : date of the observation.
- `iso_week` : week number of a given year.
- `open` : open value of a single Bitcoin coin.
- `high` : highest value achieved during a given day period.
- `low` : lowest value achieved during a given day period.
- `close` : value at the close of the transaction day.
- `volume` : what is the total volume of Bitcoin that was exchanged during that day.

- `market_capitalization` : as described in CoinMarketCap's FAQ page, this is calculated by Market Cap = Price X Circulating Supply.

All values are in USD.

## Data Exploration

The dataset timeseries is explored to understand its patterns.

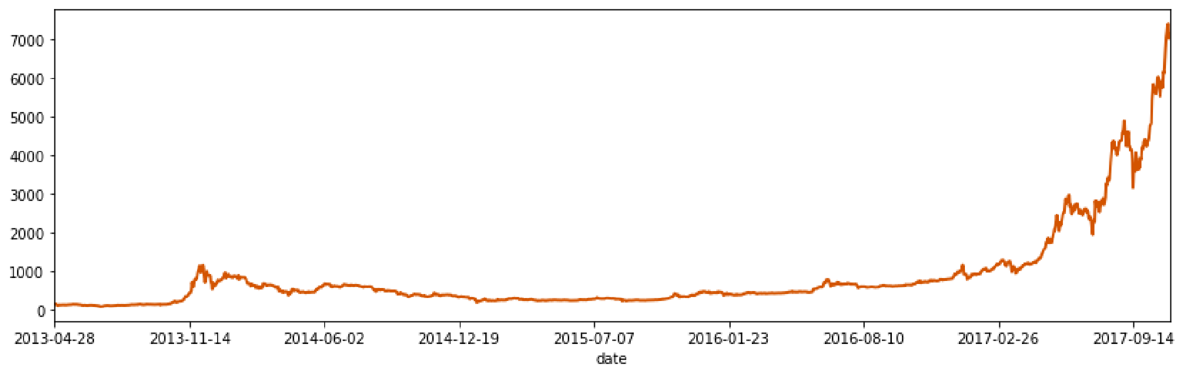
At a first instance two variables are explored: close price and volume. Volume contains data starting in November 2013, while close prices start earlier in April of the same year. However, both show similar spiking patterns starting at the beginning of 2017.

In [13]:

```
bitcoin.set_index('date')['close'].plot(linewidth=2, figsize=(14, 4), color='#d35400')
```

Out[13]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x1b878b89a20>
```

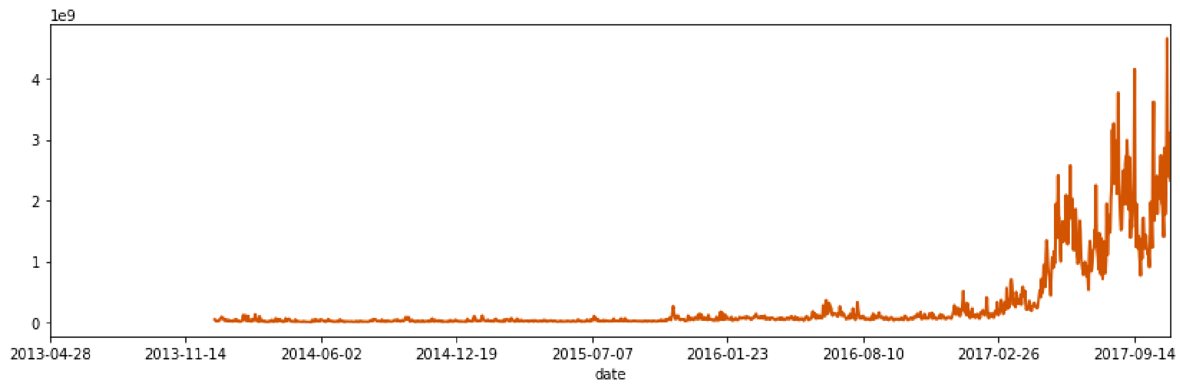


In [14]:

```
#
#  Compariosn plot for the volume variable here.

bitcoin.set_index('date')['volume'].plot(linewidth=2, figsize=(14, 4), color='#d35400')
```

Out[14]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x1b878ab3d68>
```



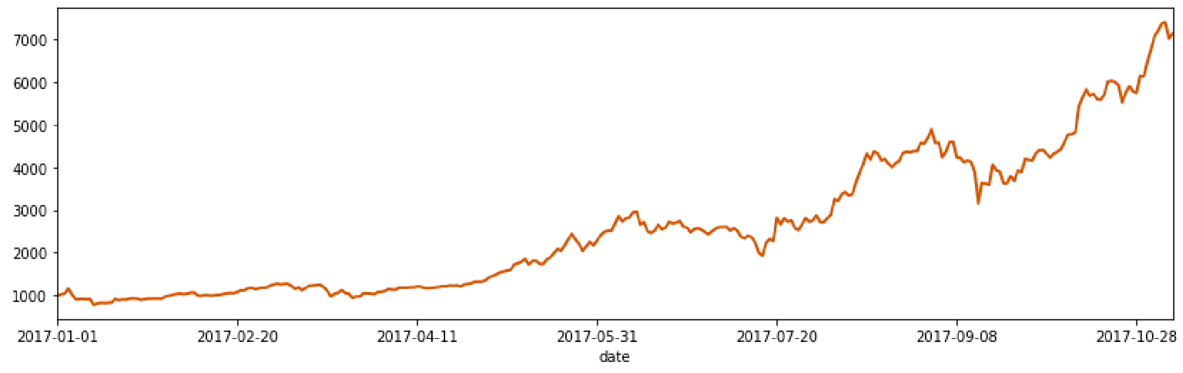
Exploration of the data of year of 2017 only. This is the year where the price of bitcoin had risen significantly.

In [15]:

```
bitcoin[bitcoin['date'] >= '2017-01-01'].set_index('date')['close'].plot(
    linewidth=2, figsize=(14, 4), color='#d35400')
```

Out[15]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x1b878b0cb38>
```

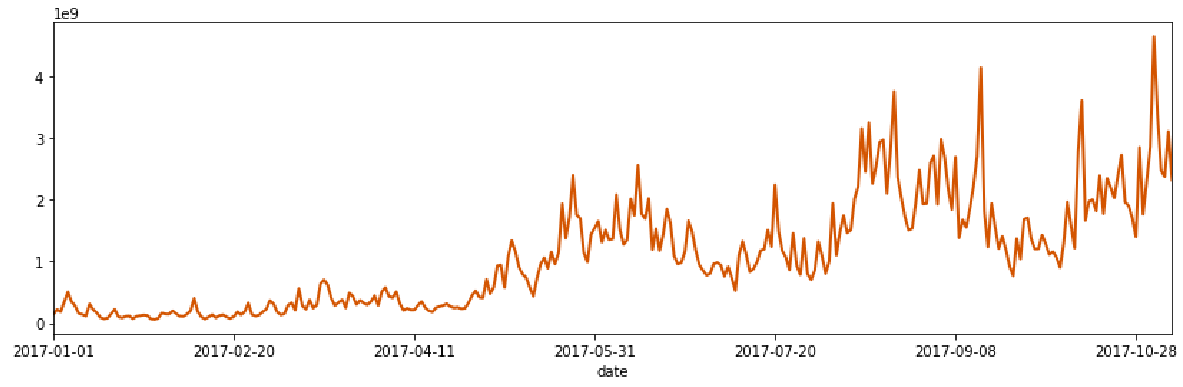


In [16]:

```
#
#  data plot for the volume variable.

bitcoin[bitcoin['date'] >= '2017-01-01'].set_index('date')['volume'].plot(
    linewidth=2, figsize=(14, 4), color='#d35400')
```

Out[16]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x1b878b4f710>
```



## Preparation of Dataset for Model

Neural networks typically work with either matrices (https://en.wikipedia.org/wiki/Matrix_(mathematics)) or tensors (https://en.wikipedia.org/wiki/Tensor). Our data needs to fit that structure before it can be used by either `keras` (or `tensorflow`).

Also, it is common practice to normalize data before using it to train a neural network. We will be using a normalization technique the evaluates each observation into a range between 0 and 1 in relation to the first observation in each week.

In [17]:

```
bitcoin.head()
```

Out[17]:

|   | date | iso_week | open | high | low | close | volume | market_capitalization |
|---|---|---|---|---|---|---|---|---|
| **0** | 2013-04-28 | 2013-17 | 135.30 | 135.98 | 132.10 | 134.21 | NaN | 1.500520e+09 |
| **1** | 2013-04-29 | 2013-17 | 134.44 | 147.49 | 134.00 | 144.54 | NaN | 1.491160e+09 |
| **2** | 2013-04-30 | 2013-17 | 144.00 | 146.93 | 134.05 | 139.00 | NaN | 1.597780e+09 |
| **3** | 2013-05-01 | 2013-17 | 139.00 | 139.89 | 107.72 | 116.99 | NaN | 1.542820e+09 |
| **4** | 2013-05-02 | 2013-17 | 116.38 | 125.60 | 92.28 | 105.21 | NaN | 1.292190e+09 |

Data was removed from older time periods keeping only the data from 2016 until the latest observation of 2017. Whereas older observations may be useful to understand current prices, Bitcoin's increased popularity in recent years makes inclusion of older data a more laborious and computationally expensive treatment which can be considered for future exploration.

In [18]:

```
bitcoin_recent = bitcoin[bitcoin['date'] >= '2016-01-01']
```

Only the "close" and "volume" variables were retained for this study.

In [27]:

```
bitcoin_recent = bitcoin_recent[['date', 'iso_week', 'close', 'volume']]
```

Next the data for both the `close` and `volume` variables were normalized

In [28]:

```
bitcoin_recent['close_point_relative_normalization'] = bitcoin_recent.groupby('iso_week')['
    lambda x: normalizations.point_relative_normalization(x))
```

In [44]:

```
#
#  The same normalization was applied on the volume variable.
bitcoin_recent['volume_point_relative_normalization'] = bitcoin_recent.groupby('iso_week')[
    lambda x: normalizations.point_relative_normalization(x))
```

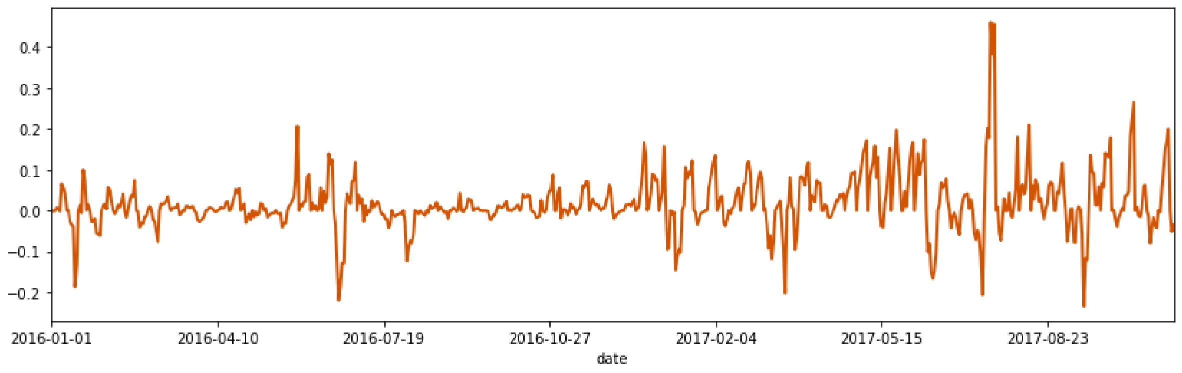
After the normalization procedure, the variables `close` and `volume` are now relative to the first observation of every week. The following variables will be used -- `close_point_relative_normalization` and `volume_point_relative_normalization` , respectivelly -- to train the LSTM model.

In [45]:

```
bitcoin_recent.set_index('date')['close_point_relative_normalization'].plot(
    linewidth=2, figsize=(14, 4), color='#d35400')
```

Out[45]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x1b879e35908>
```

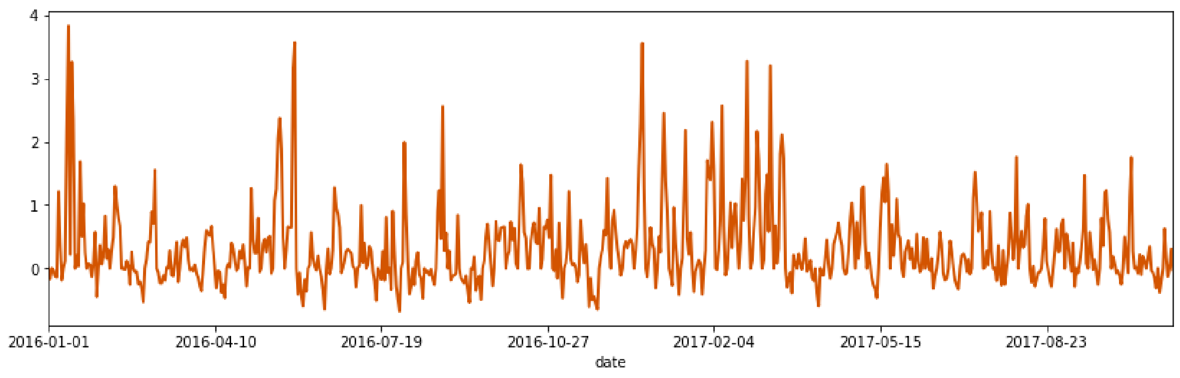


In [46]:

```
#
#  Plot of the volume variable (`volume_point_relative_normalization`)
bitcoin_recent.set_index('date')['volume_point_relative_normalization'].plot(
    linewidth=2, figsize=(14, 4), color='#d35400')
```

Out[46]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x1b879e124a8>
```



## Training and Test Sets

The dataset was divided into a training and a test set. In this case, 80% of the dataset was used to train the LSTM model and 20% to test its performance. Given that the data is continuous, last 20% of available weeks was used as a test set and the first 80% as a training set.

In [47]:

```
boundary = int(0.8 * bitcoin_recent['iso_week'].nunique())
train_set_weeks = bitcoin_recent['iso_week'].unique()[0:boundary]
test_set_weeks = bitcoin_recent[~bitcoin_recent['iso_week'].isin(train_set_weeks)]['iso_wee
```

In [48]:

```
train_set_weeks
```

Out[48]:

```
array(['2016-00', '2016-01', '2016-02', '2016-03', '2016-04', '2016-05',
       '2016-06', '2016-07', '2016-08', '2016-09', '2016-10', '2016-11',
       '2016-12', '2016-13', '2016-14', '2016-15', '2016-16', '2016-17',
       '2016-18', '2016-19', '2016-20', '2016-21', '2016-22', '2016-23',
       '2016-24', '2016-25', '2016-26', '2016-27', '2016-28', '2016-29',
       '2016-30', '2016-31', '2016-32', '2016-33', '2016-34', '2016-35',
       '2016-36', '2016-37', '2016-38', '2016-39', '2016-40', '2016-41',
       '2016-42', '2016-43', '2016-44', '2016-45', '2016-46', '2016-47',
       '2016-48', '2016-49', '2016-50', '2016-51', '2016-52', '2017-01',
       '2017-02', '2017-03', '2017-04', '2017-05', '2017-06', '2017-07',
       '2017-08', '2017-09', '2017-10', '2017-11', '2017-12', '2017-13',
       '2017-14', '2017-15', '2017-16', '2017-17', '2017-18', '2017-19',
       '2017-20', '2017-21', '2017-22', '2017-23', '2017-24', '2017-25'],
      dtype=object)
```

In [49]:

```
test_set_weeks
```

Out[49]:

```
array(['2017-26', '2017-27', '2017-28', '2017-29', '2017-30', '2017-31',
       '2017-32', '2017-33', '2017-34', '2017-35', '2017-36', '2017-37',
       '2017-38', '2017-39', '2017-40', '2017-41', '2017-42', '2017-43',
       '2017-44', '2017-45'], dtype=object)
```

Creation of separate datasets for each operation.

In [50]:

```
train_dataset = bitcoin_recent[bitcoin_recent['iso_week'].isin(train_set_weeks)]
```

In [51]:

```

#  `test_set_weeks` list to create the variable `test_dataset`.

test_dataset = bitcoin_recent[bitcoin_recent['iso_week'].isin(test_set_weeks)]
```

## Storing Output

The output is stored on disk to make sure it is easier to use this data as input to our neural network.

In [52]:

```
bitcoin_recent.to_csv('C:\Knowledgebase\Certs\DataScience\BitCoin\data/bitcoin_recent.csv',
train_dataset.to_csv('C:\Knowledgebase\Certs\DataScience\BitCoin\data/train_dataset.csv', i
test_dataset.to_csv('C:\Knowledgebase\Certs\DataScience\BitCoin\data/test_dataset.csv', inc
```

## Conclusion

In this section, the Bitcoin dataset was explored. During the year of 2017 the prices of Bitcoin skyrocketed. This phenomenon takes a long time to take place—and may have been influenced by a number of external factors that this data alone doesn't explain (for instance, the emergence of other cryptocurrencies).

However, Bitcoin statistics show an upward trend. In the section, an effort will be made to build a predictive model.

## Section 2.2.2: Predictive Model: Creating a TensorFlow Model using Keras

This section addresses design and compilation of a deep learning model using Keras as an interface to TensorFlow. This will be used as the base model in this project and will assist in experimenting with different optimization techniques. However, the essential components of the model are entirely designed in this notebook.

In [30]:

```
from keras.models import Sequential
from keras.models import load_model
from keras.layers.recurrent import LSTM
from keras.layers.core import Dense, Activation
```

### Building a Model

The dataset contains daily observations and each observation influences a future observation. Also, it is of interest to be able to predict a week--that is, seven days--of Bitcoin prices in the future. For the above reasons, the parameters `period_length` and `number_of_observations` are chosen as follows:

- `period_length` : the size of the period to use as training input. The periods are organized in distinct weeks. The model will be built using a 7-day period to predict a week in the future.
- `number_of_observations` : defines distinct periods the dataset has. There are 77 weeks available in the dataset. However, the very last week will be used to test the LSTM network on every epoch, what will be used is 77 - 1 = 76 periods for training it.

In [31]:

```
period_length = 7
number_of_periods = 76
```

Building the LSTM model.

In [32]:

```
def build_model(period_length, number_of_periods, batch_size=1):
    """
    Builds an LSTM model using Keras. This function
    works as a simple wrapper for a manually created
    model.

    Parameters
    ----------
    period_length: int
        The size of each observation used as input.

    number_of_periods: int
        The number of periods available in the
        dataset.

    batch_size: int
        The size of the batch used in each training
        period.

    Returns
    -------
    model: Keras model
        Compiled Keras model that can be trained
        and stored in disk.
    """
    model = Sequential()
    model.add(LSTM(
        units=period_length,
        batch_input_shape=(batch_size, number_of_periods, period_length),
        input_shape=(number_of_periods, period_length),
        return_sequences=False, stateful=False))

    model.add(Dense(units=period_length))
    model.add(Activation("linear"))

    model.compile(loss="mse", optimizer="rmsprop")

    return model
```

## Saving Model

The function `build_model()` is used as a starting point for building the model. That function will be refactored when building the Flask application for making it easier to train the network and use it for predictions. The model output is stored on disk for now.

In [33]:

```
model = build_model(period_length=period_length, number_of_periods=number_of_periods)
```

In [34]:

```
model.save('bitcoin_lstm_v0.h5')
```

The steps above compile the LSTM model as TensorFlow computation graph. This model can now be trained using the training set and evaluate its results with the test set.

In [35]:

```
import pandas as pd
train = pd.read_csv('C:\Knowledgebase\Certs\DataScience\BitCoin\data/train_dataset.csv')
```

In [36]:

```
train.head()
```

Out[36]:

| | date | iso_week | close | volume | close_point_relative_normalization | volume_point_relative |
|---|---|---|---|---|---|---|
| **0** | 2016-01-01 | 2016-00 | 434.33 | 36278900.0 | 0.000000 | |
| **1** | 2016-01-02 | 2016-00 | 433.44 | 30096600.0 | -0.002049 | |
| **2** | 2016-01-03 | 2016-01 | 430.01 | 39633800.0 | 0.000000 | |
| **3** | 2016-01-04 | 2016-01 | 433.09 | 38477500.0 | 0.007163 | |
| **4** | 2016-01-05 | 2016-01 | 431.96 | 34522600.0 | 0.004535 | |

LSTM networks require vectors with three dimensions. These dimensions are:

- **Period length**: The period length, i.e. how many observations is there on a period.
- **Number of periods**: How many periods are available in the dataset.
- **Number of features**: Number of features available in the dataset.

Weekly groups was craeted and then the resulting array rearranged to match those dimensions.

In [37]:

```
import matplotlib.pyplot as plt
import numpy as np
plt.style.use('seaborn-white')
```

## Shaping Data

Neural networks typically work with vectors and tensors, both mathematical objects that organize data in a number of dimensions.

In [38]:

```
# Function to create weekly groups
def create_groups(data, group_size=7):
    """
    Creates distinct groups from a given continuous series.

    Parameters
    ----------
    data: np.array
        Series of continious observations.

    group_size: int, default 7
        Determines how large the groups are. That is,
        how many observations each group contains.

    Returns
    -------
    A Numpy array object.
    """
    samples = list()
    for i in range(0, len(data), group_size):
        sample = list(data[i:i + group_size])
        if len(sample) == group_size:
            samples.append(np.array(sample).reshape(1, group_size).tolist())

    return np.array(samples)
```

In [39]:

```
data = create_groups(train['close_point_relative_normalization'].values)
```

In [40]:

```
X_train = data[:-1,:].reshape(1, 76, 7)
Y_validation = data[-1].reshape(1, 7)
```

## Load The Model

Load our previously trained model.

In [41]:

```
model = load_model('bitcoin_lstm_v0.h5')
```

## Predictions

In [42]:

```
%%time
history = model.fit(
    x=X_train, y=Y_validation,
    batch_size=32, epochs=100)
```

```
Epoch 84/100
1/1 [==============================] - 0s 19ms/step - loss: 3.2274e-06
Epoch 85/100

1/1 [==============================] - 0s 18ms/step - loss: 6.1250e-06
Epoch 86/100
1/1 [==============================] - 0s 26ms/step - loss: 5.6376e-06
Epoch 87/100
1/1 [==============================] - 0s 18ms/step - loss: 6.5591e-06
Epoch 88/100
1/1 [==============================] - 0s 19ms/step - loss: 5.8218e-06
Epoch 89/100
1/1 [==============================] - 0s 29ms/step - loss: 3.5473e-06
Epoch 90/100
1/1 [==============================] - 0s 20ms/step - loss: 2.0163e-06
Epoch 91/100
1/1 [==============================] - 0s 23ms/step - loss: 1.2039e-06
Epoch 92/100
1/1 [==============================] - 0s 23ms/step - loss: 8.2755e-07
Epoch 93/100
1/1 [==============================] - 0s 21ms/step - loss: 6.5830e-07
```

In [43]:

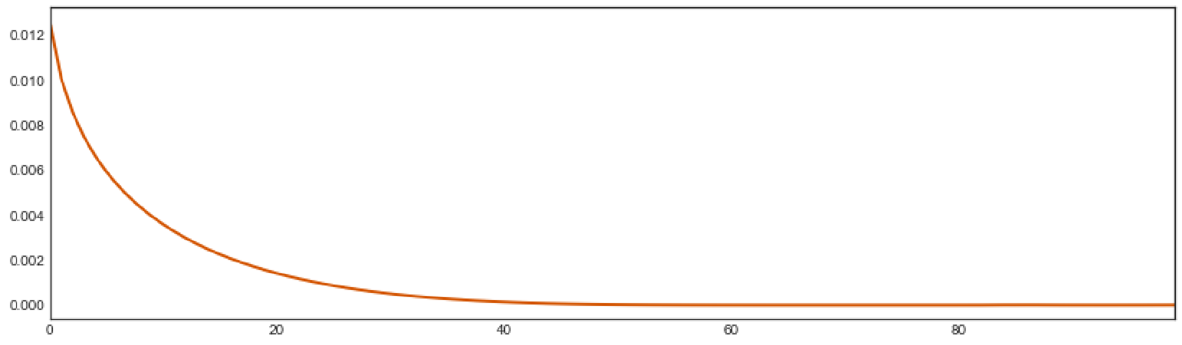
```
model.save('bitcoin_lstm_v0_trained.h5')
```

In [44]:

```
pd.Series(history.history['loss']).plot(linewidth=2, figsize=(14, 4), color='#d35400')
```

Out[44]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x1fa70c8f1d0>
```



In [45]:

```
def denormalize(series, last_value):
    result = last_value * (series + 1)
    return result
```

In [46]:

```
predictions = model.predict(x=X_train)[0]
```

In [47]:

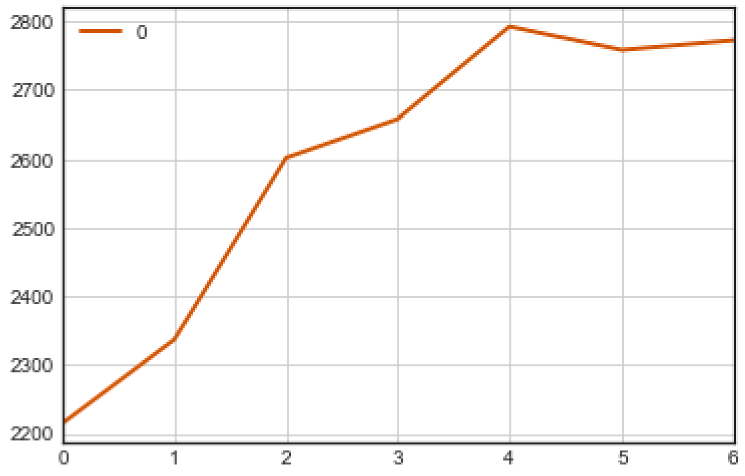
```
last_weeks_value = train[train['date'] == train['date'].max()]['close'].values[0]
denormalized_prediction = denormalize(predictions, last_weeks_value)
```

In [52]:

```
pd.DataFrame(denormalized_prediction).plot(linewidth=2, figsize=(6, 4), color='#d35400', gr
```

Out[52]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x1fa71d23710>
```

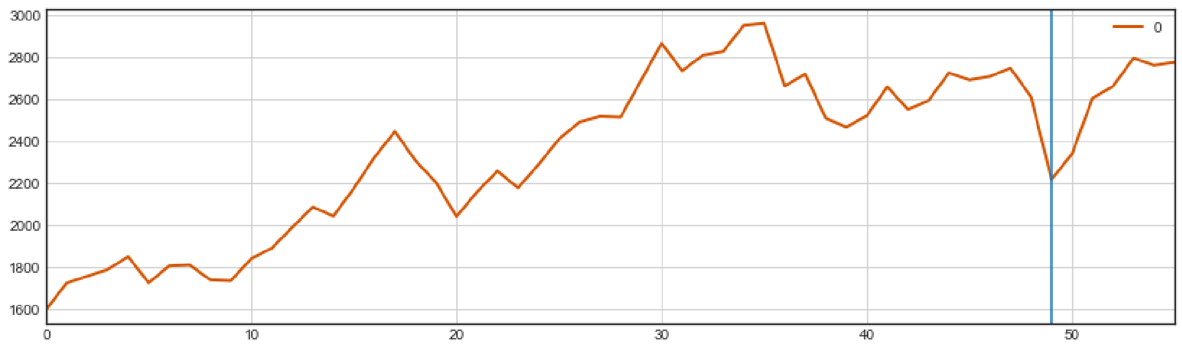


In [51]:

```
full_series = list(train['close'].values) + list(denormalized_prediction)
pd.DataFrame(full_series[-7*8:]).plot(linewidth=2, figsize=(14, 4), color='#d35400', grid=T
plt.axvline(len(full_series[-7*8:]) - 7)
```

Out[51]:

```
<matplotlib.lines.Line2D at 0x1fa71e84358>
```



## Conclusion

This section saw a complete deep learning system assembled: from data to prediction. The model created in this activity need a number of improvements before it can be considered useful. However, it serves as a good starting point from which it can continuously improve.

In [ ]:

```

```

**Magic Commands**

Magic commands (those that start with `%` ) are commands that modify a configuration of Jupyter Notebooks. A number of magic commands are available by default (see list here (http://ipython.readthedocs.io/en/stable/interactive/magics.html))--and many more can be added with extensions. The magic command added in this section allows `matplotlib` to display our plots directly on the browser instead of having to save them on a local file.

In [1]:

```
%matplotlib inline
```

## Section 2.3.1: Creating an active training environment

Evaluation of the LSTM model and training it with new data.

In [2]:

```
import math
import numpy as np
import pandas as pd
import seaborn as sb
import matplotlib.pyplot as plt
```

In [3]:

```
from keras.models import load_model
from keras.callbacks import TensorBoard
from datetime import datetime, timedelta
```

```
Using TensorFlow backend.
```

In [8]:

```
from utilities import create_groups, split_lstm_input
```

In [9]:

```
plt.style.use('seaborn-white')
```

### Validation Data

In Section 2.2, the Bitcoin dataset was separated using a 80/20 split. 80% of the data was used to train `v0` of the LSTM model and 20% was left for test purposes. These datasets are loaded into this session.

In [11]:

```
train = pd.read_csv('C:\Knowledgebase\Certs\DataScience\BitCoin\data/train_dataset.csv')
```

In [12]:

```
test = pd.read_csv('C:\Knowledgebase\Certs\DataScience\BitCoin\data/test_dataset.csv')
```

## Re-train Model with TensorBoard

The model was previously trained using a vanilla Keras implementation. Now that same model (loaded using `load_model()` ) will be retrained using the same parameters, but adding the `TensorBoard` callback. This allows us to investigate how that model is performing in near-real time.

The 'train_model()' helper function is a wrapper around the model that trains (using 'model.fit()' ) our model, storing its respective results under a new directory. Those results are then used by TensorBoard as a discriminator, in order to display statistics for different models.

In [42]:

```
def train_model(model, X, Y, epochs=100, version=0, run_number=0):
    """
    Shorthand function for training a new model.
    This function names each run of the model
    using the TensorBoard naming conventions.

    Parameters
    ----------
    model: Keras model instance
        Compiled Keras model.

    X, Y: np.array
        Series of observations to be used in
        the training process.

    epochs: int
        The number of epochs to train the
        model for.

    version: int
        Version of the model to run.

    run_number: int
        The number of the run. Used in case
        the same model version is run again.
    """
    model_name = 'bitcoin_lstm_v{version}_run_{run_number}'.format(version=version,
                                                                   run_number=run_number)

    tensorboard = TensorBoard(log_dir='./logs/{}'.format(model_name))

    model_history = model.fit(
        x=X, y=Y,
        batch_size=1, epochs=100,
        verbose=0, callbacks=[tensorboard],
        shuffle=False)

    return model_history
```

In [16]:

```
train_data = create_groups(
    train['close_point_relative_normalization'].values)
```

In [17]:

```
test_data = create_groups(
    test['close_point_relative_normalization'].values)
```

In [18]:

```
X_train, Y_train = split_lstm_input(train_data)
```

In [19]:

```
model = load_model('bitcoin_lstm_v0.h5')
```

```
WARNING:tensorflow:From C:\Users\santagan\AppData\Local\Continuum\anaconda3
\envs\qml\lib\site-packages\tensorflow\python\framework\op_def_library.py:26
3: colocate_with (from tensorflow.python.framework.ops) is deprecated and wi
ll be removed in a future version.
Instructions for updating:
Colocations handled automatically by placer.
```

In [20]:

```
model_history = train_model(model=model, X=X_train, Y=Y_train, epochs=100, version=0, run_r
```

```
WARNING:tensorflow:From C:\Users\santagan\AppData\Local\Continuum\anaconda3
\envs\qml\lib\site-packages\tensorflow\python\ops\math_ops.py:3066: to_int32
(from tensorflow.python.ops.math_ops) is deprecated and will be removed in a
future version.
Instructions for updating:
Use tf.cast instead.
```

## Evaluate LSTM Model

Evaluation of model performance against unseen data is done in this section. The model is trained in 76 weeks to predict a following weeks which is a sequence of 7 days. At the start of this project the original dataset was divided between a test and a validation set. Now take that originally trained network containing 76 weeks will be taken and usedc to predict all the 19 weeks from the validation set.

In order to do that a sequence of 76 weeks is needed as the data used for predictions. To get that data in a continuous way, the training and validation sets were combined and then 76 window was moved from the beginning of the series until its end - 1. There is an "1" at the end because that is the final target prediction that can be made.

At each one of these iterations, the LSTM model generates a 7-day prediction. Those predictions are taken and separated. Then the predicted series is compared with all the weeks in the validation set. This is done by computing both MSE and MAPE on that final series.

In [43]:

```
combined_set = np.concatenate((train_data, test_data), axis=1)
```

In [44]:

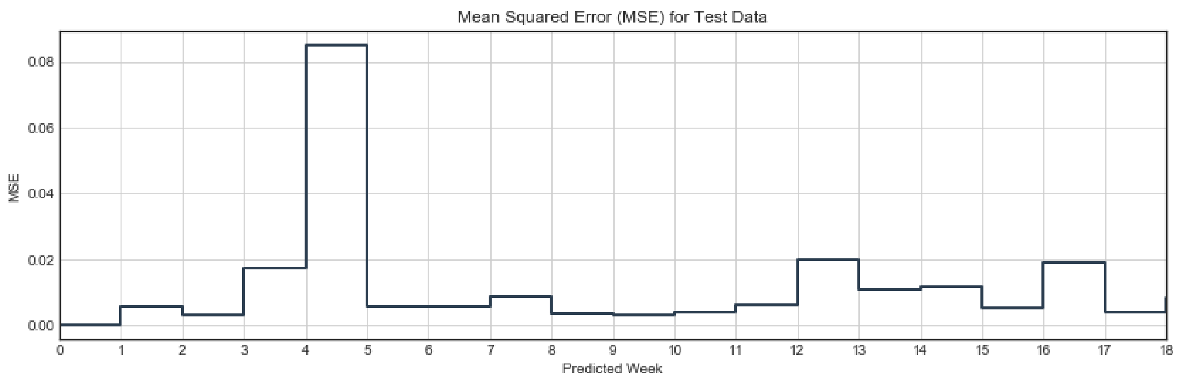
```
evaluated_weeks = []
for i in range(0, test_data.shape[1]):
    input_series = combined_set[0:,i:i+77]

    X_test = input_series[0:,:-1].reshape(1, input_series.shape[1] - 1, 7)
    Y_test = input_series[0:,-1:][0]

    result = model.evaluate(x=X_test, y=Y_test, verbose=0)
    evaluated_weeks.append(result)
```

In [23]:

```
ax = pd.Series(evaluated_weeks).plot(drawstyle="steps-post",
                                     figsize=(14,4),
                                     linewidth=2,
                                     color='#2c3e50',
                                     grid=True,
                                     title='Mean Squared Error (MSE) for Test Data')

y = [i for i in range(0, len(evaluated_weeks))]
yint = range(min(y), math.ceil(max(y))+1)
plt.xticks(yint)

ax.set_xlabel("Predicted Week")
ax.set_ylabel("MSE")
```

Out[23]:

```
Text(0, 0.5, 'MSE')
```



## Interpreting the Model Results

MSE is a good loss function for the current problem, but its results are difficult to interpret. Two utility functions are used to facilitate the interpretation of results: Root Mean Squared Error ( `rmse()` ) and Mean Absolute Error ( `mape()` ). These functions will be sued for both the observed and predicted series.

### Make Predictions

Predictions for every week of the dataset using a similar technique. Instead of calling the `evaluate()` method to compute the network's MSE, the `predict()` method is sued for making future predictions.

In [24]:

```
predicted_weeks = []
for i in range(0, test_data.shape[1]):
    input_series = combined_set[0:,i:i+76]
    predicted_weeks.append(model.predict(input_series))

predicted_days = []
for week in predicted_weeks:
    predicted_days += list(week[0])
```

Creation of a new Pandas DataFrame with the predicted values as a mean to plotting and manipulating data.

In [25]:

```
combined = pd.concat([train, test])
```

In [26]:

```
last_day = datetime.strptime(train['date'].max(), '%Y-%m-%d')
list_of_days = []
for days in range(1, len(predicted_days) + 1):
    D = (last_day + timedelta(days=days)).strftime('%Y-%m-%d')
    list_of_days.append(D)
```

In [27]:

```
predicted = pd.DataFrame({
    'date': list_of_days,
    'close_point_relative_normalization': predicted_days
})
```

In [28]:

```
combined['date'] = combined['date'].apply(
                    lambda x: datetime.strptime(x, '%Y-%m-%d'))
```

In [29]:

```
predicted['date'] = predicted['date'].apply(lambda x: datetime.strptime(x, '%Y-%m-%d'))
```

In [30]:

```
observed = combined[combined['date'] > train['date'].max()]
```

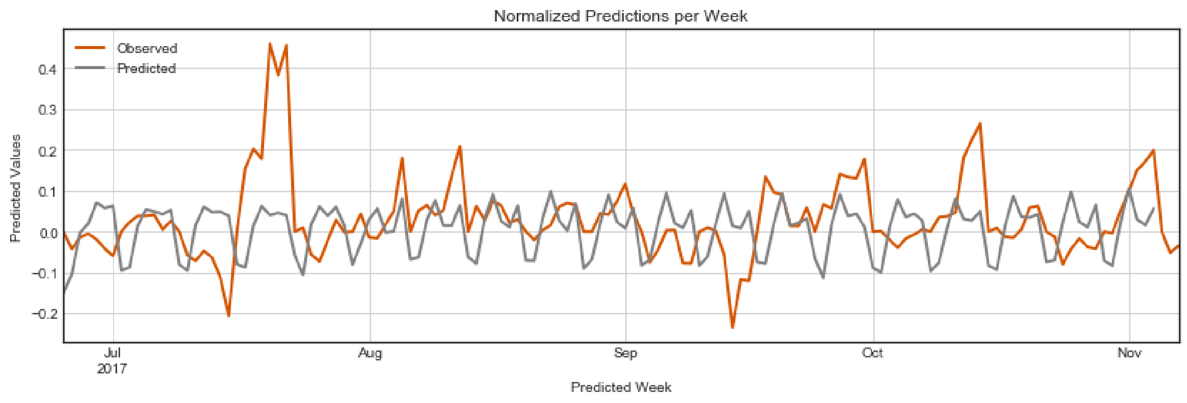
The graph below compares the predicted value versus the real one.

In [31]:

```
def plot_two_series(A, B, variable, title):
    """
    Plots two series using the same `date` index.

    Parameters
    ----------
    A, B: pd.DataFrame
        Dataframe with a `date` key and a variable
        passed in the `variable` parameter. Parameter A
        represents the "Observed" series and B the "Predicted"
        series. These will be labelled respectivelly.

    variable: str
        Variable to use in plot.

    title: str
        Plot title.

    """
    plt.figure(figsize=(14,4))
    plt.xlabel('Real and predicted')

    ax1 = A.set_index('date')[variable].plot(
        linewidth=2, color='#d35400', grid=True, label='Observed', title=title)

    ax2 = B.set_index('date')[variable].plot(
        linewidth=2, color='grey', grid=True, label='Predicted')

    ax1.set_xlabel("Predicted Week")
    ax1.set_ylabel("Predicted Values")

    h1, l1 = ax1.get_legend_handles_labels()
    h2, l2 = ax2.get_legend_handles_labels()

    plt.legend(l1+l2, loc=2)
    plt.show()
```

In [32]:

```
plot_two_series(observed, predicted,
                variable='close_point_relative_normalization',
                title='Normalized Predictions per Week')
```



## De-normalized Predictions

Comparison of the predictions with the denormalized series.

In [33]:

```
predicted['iso_week'] = predicted['date'].apply(
                            lambda x: x.strftime('%Y-%U'))
```

In [34]:

```
def denormalize(reference, series,
                normalized_variable='close_point_relative_normalization',
                denormalized_variable='close'):
    """
    Denormalizes the values for a given series.

    Parameters
    ----------
    reference: pd.DataFrame
        DataFrame to use as reference. This dataframe
        contains both a week index and the USD price
        reference that we are interested on.

    series: pd.DataFrame
        DataFrame with the predicted series. The
        DataFrame must have the same columns as the
        `reference` dataset.

    normalized_variable: str, default 'close_point_relative_normalization'
        Variable to use in normalization.

    denormalized_variable: str, `close`
        Variable to use in de-normalization.

    Returns
    -------
    A modified DataFrame with the new variable provided
    in `denormalized_variable` parameter.
    """
    week_values = observed[
        reference['iso_week'] == series['iso_week'].values[0]]
    last_value = week_values[denormalized_variable].values[0]
    series[denormalized_variable] = last_value * (series[normalized_variable] + 1)

    return series
```

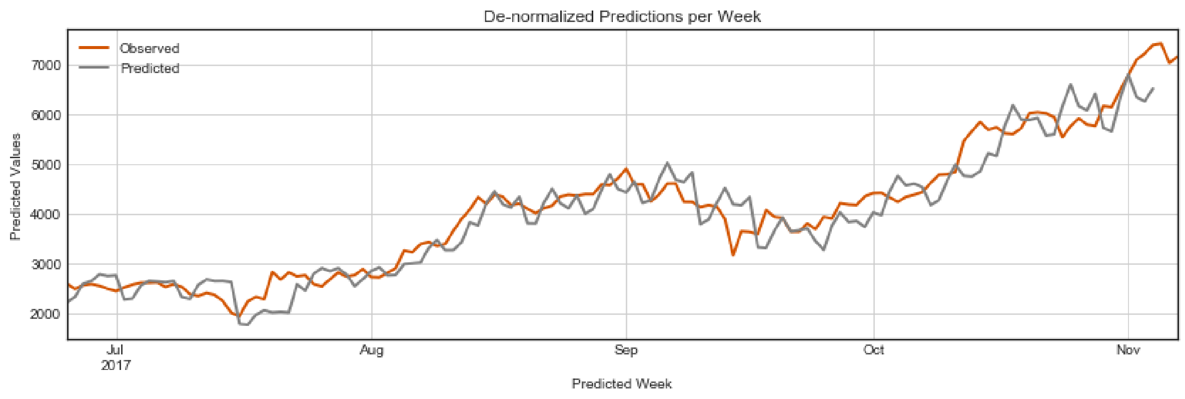
In [35]:

```
predicted_close = predicted.groupby('iso_week').apply(
                        lambda x: denormalize(observed, x))
```

Predicted timeseries versus the observed one using the `close` price values.

In [36]:

```
plot_two_series(observed, predicted_close,
                variable='close',
                title='De-normalized Predictions per Week')
```



## Calculation of RMSE and MAPE

After computing both timeseries prediction of how close the predictions are to the real, observed data is desirable. This was done by using a modified version of the loss function (the Root Mean Squared Error instead of the Mean Squared Error) and the Mean Absolute Percentage Error (MAPE) was added for readability.

In [39]:

```
from utilities import rmse, mape
```

In [40]:

```
print('Normalized RMSE: {:.2f}'.format(
    rmse(observed['close_point_relative_normalization'][:-3],
         predicted_close['close_point_relative_normalization'])))
```

```
Normalized RMSE: 0.11
```

In [41]:

```
print('De-normalized RMSE: ${:.1f} USD'.format(
    rmse(observed['close'][:-3],
         predicted_close['close'])))

print('De-normalized MAPE: {:.1f}%'.format(
    mape(observed['close'][:-3],
         predicted_close['close'])))
```

```
De-normalized RMSE: $392.5 USD
De-normalized MAPE: 8.1%
```

# Appendix 2

## Wisconsin Breast Cancer Data Analysis

The dataset for this part of the Project is taken from the publicly avaialble Wisconsin Breast Cancer database, created by Dr. William H. Wolberg, a physician at the University of Wisconsin Hospital at Madison, Wisconsin, USA. This dataset contains 569 samples of malignant and benign tumor cells. Dr. Wolberg used fluid samples, taken from patients with solid breast masses and an easy-to-use graphical computer program called Xcyt, which is capable of performing the analysis of cytological features based on a digital scan. The program uses a curve-fitting algorithm to compute ten features from each one of the cells in the sample and then calculates the mean value, extreme value and standard error of each feature for the image.

The first two columns in the dataset store the unique ID numbers of the samples and the corresponding diagnoses (M = malignant, B = benign), respectively. Columns 3-32 contain 30 real-valued features that have been computed from digitized images of the cell nuclei, which can be used to build a model to predict whether a tumor is benign or malignant.

The Breast Cancer Wisconsin dataset resides in the UCI Machine Learning Repository, and more information about this dataset can be found at https://archive.ics.uci.edu/ml/datasets/Breast+Cancer+Wisconsin+(Diagnostic) (https://archive.ics.uci.edu/ml/datasets/Breast+Cancer+Wisconsin+(Diagnostic)).

## Objectives

This work aims to analyse which features are most helpful in predicting malignant or benign cancer and to investigate general trends that may aid in model selection and hyper parameter selection. To achieve this, machine learning classification methods have been used to fit functions that can predict the discrete class of new input.

### 1. Data Exploration

Read the data and split it into training and test datasets

In [74]:

```
## Ignore Jupyter Warnings

from warnings import simplefilter
simplefilter(action='ignore', category=FutureWarning)

## Get the Libraries

from IPython.display import Image
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

## Load the dataset

data = pd.read_csv('https://archive.ics.uci.edu/ml/'
                 'machine-learning-databases'
                 '/breast-cancer-wisconsin/wdbc.data', header=None)
```

In [75]:

```
## Examine the dataset using pandas head() method

data.head()
```

Out[75]:

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | ... | 22 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **0** | 842302 | M | 17.99 | 10.38 | 122.80 | 1001.0 | 0.11840 | 0.27760 | 0.3001 | 0.14710 | ... | 25.38 | 17. |
| **1** | 842517 | M | 20.57 | 17.77 | 132.90 | 1326.0 | 0.08474 | 0.07864 | 0.0869 | 0.07017 | ... | 24.99 | 23. |
| **2** | 84300903 | M | 19.69 | 21.25 | 130.00 | 1203.0 | 0.10960 | 0.15990 | 0.1974 | 0.12790 | ... | 23.57 | 25. |
| **3** | 84348301 | M | 11.42 | 20.38 | 77.58 | 386.1 | 0.14250 | 0.28390 | 0.2414 | 0.10520 | ... | 14.91 | 26. |
| **4** | 84358402 | M | 20.29 | 14.34 | 135.10 | 1297.0 | 0.10030 | 0.13280 | 0.1980 | 0.10430 | ... | 22.54 | 16. |

5 rows × 32 columns

In [76]:

```
## Diemnsion of the dataset using "shape" attribute of python

print("Breast Cancer dataset dimensions : {}".format(data.shape))
```

```
Breast Cancer dataset dimensions : (569, 32)
```

In [77]:

```
## print the keys (Bunch object):
data.keys()
```

Out[77]:

```
Int64Index([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15,
16,
            17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31],
           dtype='int64')
```

In [78]:

```
## Check for any missing or null data points of the data set

data.isnull().sum()
```

Out[78]:

```
0     0
1     0
2     0
3     0
4     0
5     0
6     0
7     0
8     0
9     0
10    0
11    0
12    0
13    0
14    0
15    0
16    0
17    0
18    0
19    0
20    0
21    0
22    0
23    0
24    0
25    0
26    0
27    0
28    0
29    0
30    0
31    0
dtype: int64
```

## ### 2. Data Categorisation

It is observed that the datset contains 569 rows and 32 columns. The 30 features of the datset are assigned to a NumPy array X - using a LabelEncoder object from sklearn libraries, the class labels are transformed from their original string representations of 'M' and 'B' into integers:

In [79]:

```
from sklearn.preprocessing import LabelEncoder

X = data.loc[:, 2:].values
y = data.loc[:, 1].values
le = LabelEncoder()
y = le.fit_transform(y)
le.classes_
```

Out[79]:

```
array(['B', 'M'], dtype=object)
```

The class labels (diagnosis) are encoded in an array y. Value of 1 indicates the cancer is malignant and 0 means benign. The malignant tumors are represented as class 1 and the benign tumors are represented as class 0, respectively.

#### ***The mapping is verified by calling the transform method of the fitted LabelEncoder on two dummy class labels:***

In [80]:

```
le.transform(['M', 'B'])
```

Out[80]:

```
array([1, 0], dtype=int64)
```

#### **The dataset is split into a training dataset with 80 percent of the data and a separate test dataset with 20 percent of the data:**

In [81]:

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test =     train_test_split(X, y,
                     test_size=0.20,
                     stratify=y,
                     random_state=1)
```

## 3. Feature Scaling

In most cases, datasets used will contain features which vary in magnitudes, units and range. However, since most of the machine learning algorithms use Eucledian distance between two data points in their computations, it is required to get all features to the same level of magnitude. This can be achieved by scaling. This means transforming the data so that it fits within a specific scale such as 0–100 or 0–1.

Learning algorithms require input features on the same scale for optimal performance. Hence, the columns in the Breast Cancer Wisconsin dataset need to be standardized before they can be fed into a linear classifier. An assumption is made to compress the data from the initial 30 dimensions onto a lower two-dimensional subspace via Principal Component Analysis (PCA), a feature extraction technique for dimensionality reduction.

To avoid separate steps for the fitting and transformation for the training and test datasets, the StandardScaler, PCA, and LogisticRegression objects are chained in a pipeline:

In [82]:

```python
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import make_pipeline

pipe_lr = make_pipeline(StandardScaler(),
                        PCA(n_components=2),
                        LogisticRegression(random_state=1))

pipe_lr.fit(X_train, y_train)
y_pred = pipe_lr.predict(X_test)
print('Test Accuracy: %.3f' % pipe_lr.score(X_test, y_test))
```

```
Test Accuracy: 0.956
```

In the above code snippet, the make_pipeline function takes an arbitrary number of scikit-learn transformers (objects that support the fit and transform methods as input), followed by a scikit-learn estimator that implements the fit and predict methods. In the preceding code there are two transformers, StandardScaler and PCA and a LogisticRegression estimator as inputs to the make_pipeline function, which constructs a scikit-learn Pipeline object from these objects.

The scikit-learn Pipeline can be thought of as a wrapper around those individual transformers and estimators. If the fit method of Pipeline is called, the data gets passed down a series of transformers via fit and transform calls on these intermediate steps until it arrives at the estimator object (the final element in a pipeline). The estimator then gets fitted to the transformed training data. The Pipeline object takes a list of tuples as input, where the first value in each tuple is an arbitrary identifier string that can be used to access the individual elements in the pipeline.

Once the fit method was executed, StandardScaler first performed fit and transform calls on the training data. Second, the transformed training data was passed on to the next object in the pipeline, PCA. Similar to the previous step, PCA also executed fit and transform on the scaled input data and passed it to the final element of the pipeline, the estimator. Finally, the LogisticRegression estimator was fit to the training data after it underwent transformations via StandardScaler and PCA. There is no limit to the number of intermediate steps in a pipeline; however, the last pipeline element has to be an estimator.

## Using k-fold cross validation to assess model performance

One of the key steps in building a machine learning model is to estimate its performance on data that the model hasn't seen before. To find an acceptable bias-variance trade-off, the model needs to be evaluated carefully.

In k-fold cross-validation, the training dataset is randomly split into k folds without replacement, where k — 1 folds are used for the model training, and one fold is used for performance evaluation.

A slight improvement over the standard k-fold cross-validation approach is stratified k-fold cross-validation, which can yield better bias and variance estimates, especially in cases of unequal class proportions, as has been shown in a study by Ron Kohavi [9]. The code for the implementation is shown below.

At first, the StratifiedKfold iterator from the sklearn.model_selection module with the y_train class labels in the training set, with a specified number of folds via the n_splits parameter. When the kfold iterator was used to loop through the k-folds, the returned indices helped to train to fit the logistic regression pipeline. Using the pipe_lr pipeline, it was ensured that the samples were scaled properly (for instance, standardized) in each iteration. Then the test indices were used to calculate the accuracy score of the model, which was collected in the scores list to calculate the average accuracy and the standard deviation of the estimate.

In [83]:

```
import numpy as np
from sklearn.model_selection import StratifiedKFold


kfold = StratifiedKFold(n_splits=10,
                        random_state=1).split(X_train, y_train)

scores = []
for k, (train, test) in enumerate(kfold):
    pipe_lr.fit(X_train[train], y_train[train])
    score = pipe_lr.score(X_train[test], y_train[test])
    scores.append(score)
    print('Fold: %2d, Class dist.: %s, Acc: %.3f' % (k+1,
          np.bincount(y_train[train]), score))

print('\nCV accuracy: %.3f +/- %.3f' % (np.mean(scores), np.std(scores)))
```

```
Fold:  1, Class dist.: [256 153], Acc: 0.935
Fold:  2, Class dist.: [256 153], Acc: 0.935
Fold:  3, Class dist.: [256 153], Acc: 0.957
Fold:  4, Class dist.: [256 153], Acc: 0.957
Fold:  5, Class dist.: [256 153], Acc: 0.935
Fold:  6, Class dist.: [257 153], Acc: 0.956
Fold:  7, Class dist.: [257 153], Acc: 0.978
Fold:  8, Class dist.: [257 153], Acc: 0.933
Fold:  9, Class dist.: [257 153], Acc: 0.956
Fold: 10, Class dist.: [257 153], Acc: 0.956

CV accuracy: 0.950 +/- 0.014
```

#### k-fold cross validation scorer

Although the previous code example was useful to illustrate how k-fold cross-validation works, scikit-learn also implements a k-fold cross-validation scorer, which allows for evaluation of the model using stratified k-fold cross-validation:

In [84]:

```
from sklearn.model_selection import cross_val_score

scores = cross_val_score(estimator=pipe_lr,
                         X=X_train,
                         y=y_train,
                         cv=10,
                         n_jobs=1)
print('CV accuracy scores: %s' % scores)
print('CV accuracy: %.3f +/- %.3f' % (np.mean(scores), np.std(scores)))
```

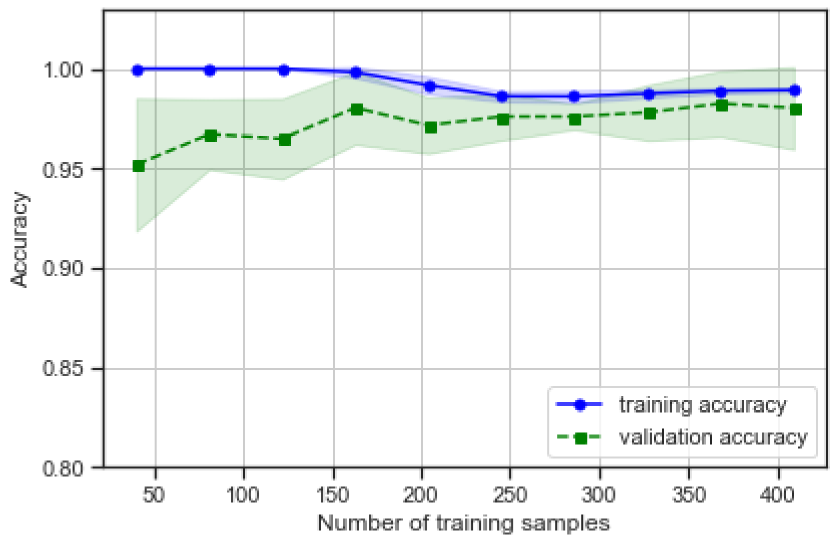
```
CV accuracy scores: [0.93478261 0.93478261 0.95652174 0.95652174 0.93478261
0.95555556
 0.97777778 0.93333333 0.95555556 0.95555556]
CV accuracy: 0.950 +/- 0.014
```

## Debugging algorithms with learning curves

### Diagnosing bias and variance problems with learning curves

In [85]:

```
import matplotlib.pyplot as plt
from sklearn.model_selection import learning_curve


pipe_lr = make_pipeline(StandardScaler(),
                        LogisticRegression(penalty='l2', random_state=1))

train_sizes, train_scores, test_scores =                learning_curve(estimator=pipe_lr,
                               X=X_train,
                               y=y_train,
                               train_sizes=np.linspace(0.1, 1.0, 10),
                               cv=10,
                               n_jobs=1)

train_mean = np.mean(train_scores, axis=1)
train_std = np.std(train_scores, axis=1)
test_mean = np.mean(test_scores, axis=1)
test_std = np.std(test_scores, axis=1)

plt.plot(train_sizes, train_mean,
         color='blue', marker='o',
         markersize=5, label='training accuracy')

plt.fill_between(train_sizes,
                 train_mean + train_std,
                 train_mean - train_std,
                 alpha=0.15, color='blue')

plt.plot(train_sizes, test_mean,
         color='green', linestyle='--',
         marker='s', markersize=5,
         label='validation accuracy')

plt.fill_between(train_sizes,
                 test_mean + test_std,
                 test_mean - test_std,
                 alpha=0.15, color='green')

plt.grid()
plt.xlabel('Number of training samples')
plt.ylabel('Accuracy')
plt.legend(loc='lower right')
plt.ylim([0.8, 1.03])
plt.tight_layout()
#plt.savefig('images/06_05.png', dpi=300)
plt.show()
```



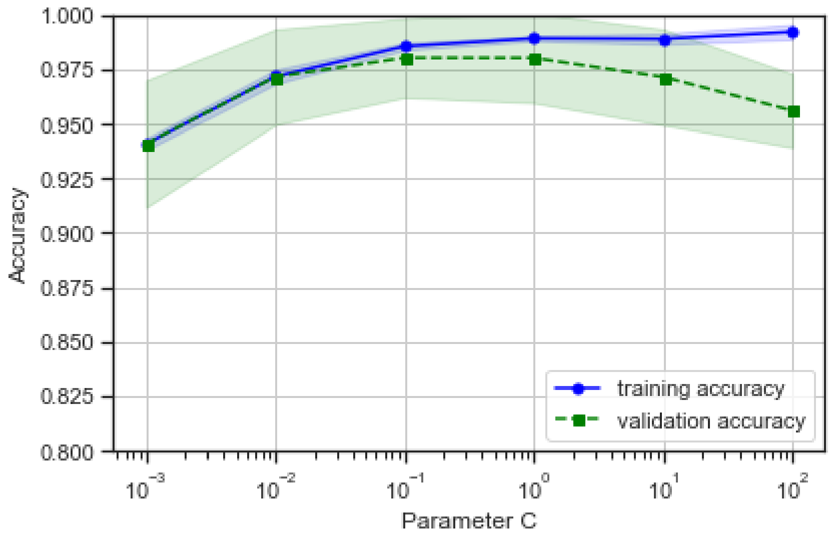
In the preceding learning curve plot, the model performs quite well on both the training and validation dataset if it had seen more than 250 samples during training. It is also apparent that the training accuracy increases for training sets with fewer than 250 samples, and the gap between validation and training accuracy widens—an indicator of an increasing degree of overfitting. The relatively small, but visible gap between training and cross-validation curves is an indicator of slight overfitting.

**Addressing over- and underfitting with validation curves**

Validation curves are a useful tool for improving the performance of a model by addressing issues such as overfitting or underfitting. Validation curves are related to learning curves, but instead of plotting the training and test accuracies as functions of the sample size, the values of the model parameters are varied. In this code, the inverse regularization parameter C in logistic regression is tweaked.

In [86]:

```
from sklearn.model_selection import validation_curve


param_range = [0.001, 0.01, 0.1, 1.0, 10.0, 100.0]
train_scores, test_scores = validation_curve(
                estimator=pipe_lr,
                X=X_train,
                y=y_train,
                param_name='logisticregression__C',
                param_range=param_range,
                cv=10)

train_mean = np.mean(train_scores, axis=1)
train_std = np.std(train_scores, axis=1)
test_mean = np.mean(test_scores, axis=1)
test_std = np.std(test_scores, axis=1)

plt.plot(param_range, train_mean,
         color='blue', marker='o',
         markersize=5, label='training accuracy')

plt.fill_between(param_range, train_mean + train_std,
                 train_mean - train_std, alpha=0.15,
                 color='blue')

plt.plot(param_range, test_mean,
         color='green', linestyle='--',
         marker='s', markersize=5,
         label='validation accuracy')

plt.fill_between(param_range,
                 test_mean + test_std,
                 test_mean - test_std,
                 alpha=0.15, color='green')

plt.grid()
plt.xscale('log')
plt.legend(loc='lower right')
plt.xlabel('Parameter C')
plt.ylabel('Accuracy')
plt.ylim([0.8, 1.0])
plt.tight_layout()
# plt.savefig('images/06_06.png', dpi=300)
plt.show()
```



## Tuning hyperparameters via grid search

Grid search is a brute-force exhaustive search paradigm where a list of values are specified for different hyperparameters, and the computer evaluates the model performance for each combination of those to obtain the optimal combination of values from this set.

In [87]:

```
from sklearn.model_selection import GridSearchCV
from sklearn.svm import SVC

pipe_svc = make_pipeline(StandardScaler(),
                         SVC(random_state=1))

param_range = [0.0001, 0.001, 0.01, 0.1, 1.0, 10.0, 100.0, 1000.0]

param_grid = [{'svc__C': param_range,
               'svc__kernel': ['linear']},
              {'svc__C': param_range,
               'svc__gamma': param_range,
               'svc__kernel': ['rbf']}]

gs = GridSearchCV(estimator=pipe_svc,
                  param_grid=param_grid,
                  scoring='accuracy',
                  cv=10,
                  n_jobs=-1)
gs = gs.fit(X_train, y_train)
print(gs.best_score_)
print(gs.best_params_)
```

```
0.9846153846153847
{'svc__C': 100.0, 'svc__gamma': 0.001, 'svc__kernel': 'rbf'}
```

In [88]:

```
clf = gs.best_estimator_
clf.fit(X_train, y_train)
print('Test accuracy: %.3f' % clf.score(X_test, y_test))
```

```
Test accuracy: 0.974
```

## Algorithm selection with nested cross-validation

In [89]:

```
gs = GridSearchCV(estimator=pipe_svc,
                  param_grid=param_grid,
                  scoring='accuracy',
                  cv=2)

scores = cross_val_score(gs, X_train, y_train,
                         scoring='accuracy', cv=5)
print('CV accuracy: %.3f +/- %.3f' % (np.mean(scores),
                                      np.std(scores)))
```

```
CV accuracy: 0.974 +/- 0.015
```

In [90]:

```
from sklearn.tree import DecisionTreeClassifier

gs = GridSearchCV(estimator=DecisionTreeClassifier(random_state=0),
                  param_grid=[{'max_depth': [1, 2, 3, 4, 5, 6, 7, None]}],
                  scoring='accuracy',
                  cv=2)

scores = cross_val_score(gs, X_train, y_train,
                         scoring='accuracy', cv=5)
print('CV accuracy: %.3f +/- %.3f' % (np.mean(scores),
                                      np.std(scores)))
```

```
CV accuracy: 0.934 +/- 0.016
```

### Reading a confusion matrix

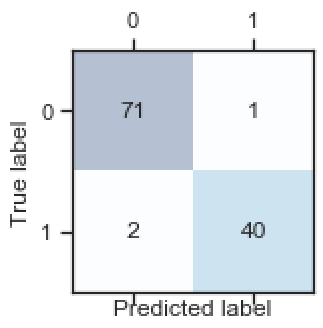
In [91]:

```
from sklearn.metrics import confusion_matrix

pipe_svc.fit(X_train, y_train)
y_pred = pipe_svc.predict(X_test)
confmat = confusion_matrix(y_true=y_test, y_pred=y_pred)
print(confmat)
```

```
[[71  1]
 [ 2 40]]
```

In [92]:

```
fig, ax = plt.subplots(figsize=(2.5, 2.5))
ax.matshow(confmat, cmap=plt.cm.Blues, alpha=0.3)
for i in range(confmat.shape[0]):
    for j in range(confmat.shape[1]):
        ax.text(x=j, y=i, s=confmat[i, j], va='center', ha='center')

plt.xlabel('Predicted label')
plt.ylabel('True label')

plt.tight_layout()
## plt.savefig('images/06_09.png', dpi=300)
plt.show()
```



**Previously the class labels were encoded so that *malignant* samples are the "postive" class (1), and *benign* samples are the "negative" class (0):**

In [93]:

```
le.transform(['M', 'B'])
```

Out[93]:

```
array([1, 0], dtype=int64)
```

In [94]:

```
confmat = confusion_matrix(y_true=y_test, y_pred=y_pred)
print(confmat)
```

```
[[71  1]
 [ 2 40]]
```

In [95]:

```
## print confusion matrix

confmat = confusion_matrix(y_true=y_test, y_pred=y_pred)
print(confmat)
```

```
[[71  1]
 [ 2 40]]
```

**Note that the (true) class 0 samples that are correctly predicted as class 0 (true negatives) are now in**

**the upper left corner of the matrix (index 0, 0). In order to change the ordering so that the true negatives are in the lower right corner (index 1,1) and the true positves are in the upper left, we can use the `labels` argument like shown below:**

In [96]:

```
confmat = confusion_matrix(y_true=y_test, y_pred=y_pred, labels=[1, 0])
print(confmat)
```

```
[[40  2]
 [ 1 71]]
```

## Conclusion:

Assuming that class 1 (malignant) is the positive class in this example, ther model correctly classified 71 of the samples that belong to class 0 (true negatives) and 40 samples that belong to class 1 (true positives), respectively. However, the model also incorrectly misclassified 1 sample from class 0 as class 1 (false positive), and it predicted that 2 samples are benign although it is a malignant tumor (false negatives).

## Optimizing the precision and recall of a classification model

In [97]:

```
from sklearn.metrics import precision_score, recall_score, f1_score

print('Precision: %.3f' % precision_score(y_true=y_test, y_pred=y_pred))
print('Recall: %.3f' % recall_score(y_true=y_test, y_pred=y_pred))
print('F1: %.3f' % f1_score(y_true=y_test, y_pred=y_pred))
```

```
Precision: 0.976
Recall: 0.952
F1: 0.964
```

In [98]:

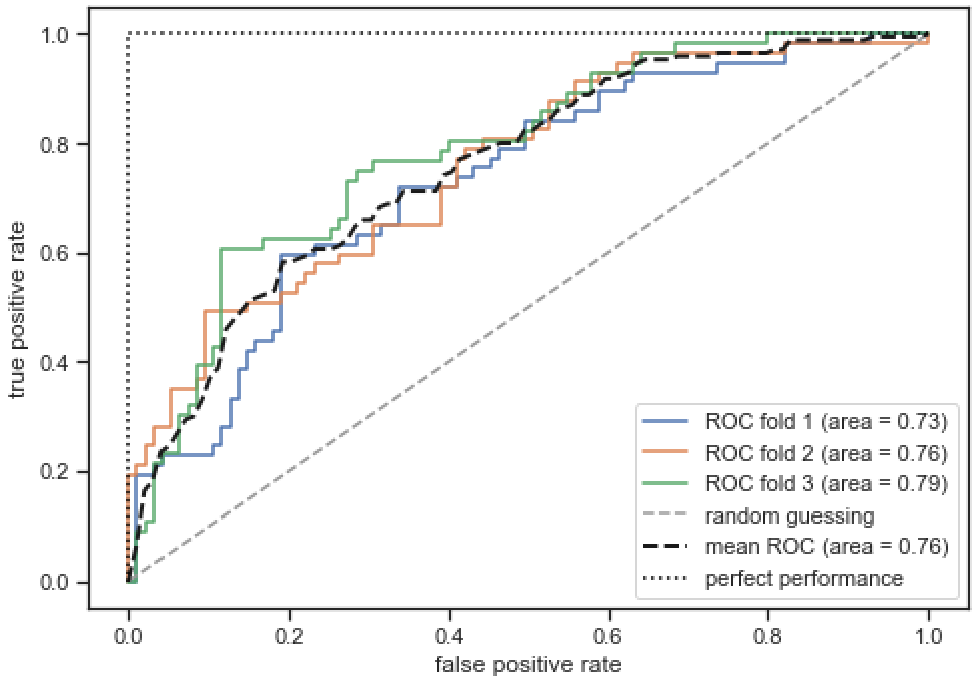
```
from sklearn.metrics import make_scorer

scorer = make_scorer(f1_score, pos_label=0)

c_gamma_range = [0.01, 0.1, 1.0, 10.0]

param_grid = [{'svc__C': c_gamma_range,
               'svc__kernel': ['linear']},
              {'svc__C': c_gamma_range,
               'svc__gamma': c_gamma_range,
               'svc__kernel': ['rbf']}]

gs = GridSearchCV(estimator=pipe_svc,
                  param_grid=param_grid,
                  scoring=scorer,
                  cv=10,
                  n_jobs=-1)
gs = gs.fit(X_train, y_train)
print(gs.best_score_)
print(gs.best_params_)
```

```
0.9862021456964396
{'svc__C': 10.0, 'svc__gamma': 0.01, 'svc__kernel': 'rbf'}
```

## Receiver Operating Characteristic (ROC)

In [99]:

```
from sklearn.metrics import roc_curve, auc
from scipy import interp

pipe_lr = make_pipeline(StandardScaler(),
                        PCA(n_components=2),
                        LogisticRegression(penalty='l2',
                                           random_state=1,
                                           C=100.0))

X_train2 = X_train[:, [4, 14]]


cv = list(StratifiedKFold(n_splits=3,
                          random_state=1).split(X_train, y_train))

fig = plt.figure(figsize=(7, 5))

mean_tpr = 0.0
mean_fpr = np.linspace(0, 1, 100)
all_tpr = []

for i, (train, test) in enumerate(cv):
    probas = pipe_lr.fit(X_train2[train],
                         y_train[train]).predict_proba(X_train2[test])

    fpr, tpr, thresholds = roc_curve(y_train[test],
                                     probas[:, 1],
                                     pos_label=1)
    mean_tpr += interp(mean_fpr, fpr, tpr)
    mean_tpr[0] = 0.0
    roc_auc = auc(fpr, tpr)
    plt.plot(fpr,
             tpr,
             label='ROC fold %d (area = %0.2f)'
                   % (i+1, roc_auc))

plt.plot([0, 1],
         [0, 1],
         linestyle='--',
         color=(0.6, 0.6, 0.6),
         label='random guessing')

mean_tpr /= len(cv)
mean_tpr[-1] = 1.0
mean_auc = auc(mean_fpr, mean_tpr)
plt.plot(mean_fpr, mean_tpr, 'k--',
         label='mean ROC (area = %0.2f)' % mean_auc, lw=2)
plt.plot([0, 0, 1],
         [0, 1, 1],
         linestyle=':',
         color='black',
         label='perfect performance')

plt.xlim([-0.05, 1.05])
plt.ylim([-0.05, 1.05])
plt.xlabel('false positive rate')
plt.ylabel('true positive rate')
plt.legend(loc="lower right")
plt.tight_layout()
# plt.savefig('images/06_10.png', dpi=300)
plt.show()
```



## The scoring metrics for multiclass classification

In [100]:

```
pre_scorer = make_scorer(score_func=precision_score,
                         pos_label=1,
                         greater_is_better=True,
                         average='micro')
```

## Dealing with class imbalance

In [101]:

```
X_imb = np.vstack((X[y == 0], X[y == 1][:40]))
y_imb = np.hstack((y[y == 0], y[y == 1][:40]))
```

In [102]:

```
y_pred = np.zeros(y_imb.shape[0])
np.mean(y_pred == y_imb) * 100
```

Out[102]:

```
89.92443324937027
```

The scikit-learn library implements a simple resample function that can help with the up-sampling of the minority class by drawing new samples from the dataset with replacement. The following code takes the minority class from the imbalanced breast cancer dataset (class 1) and repeatedly draw new samples from it until it contains the same number of samples as class label 0:

In [103]:

```
from sklearn.utils import resample

print('Number of class 1 samples before:', X_imb[y_imb == 1].shape[0])

X_upsampled, y_upsampled = resample(X_imb[y_imb == 1],
                                    y_imb[y_imb == 1],
                                    replace=True,
                                    n_samples=X_imb[y_imb == 0].shape[0],
                                    random_state=123)

print('Number of class 1 samples after:', X_upsampled.shape[0])
```

```
Number of class 1 samples before: 40
Number of class 1 samples after: 357
```

In [104]:

```
X_bal = np.vstack((X[y == 0], X_upsampled))
y_bal = np.hstack((y[y == 0], y_upsampled))
```

In [105]:

```
y_pred = np.zeros(y_bal.shape[0])
np.mean(y_pred == y_bal) * 100
```

Out[105]:

```
50.0
```

# Wisconsin Breast Cancer Data Analysis

The dataset for this part of the Project is taken from the publicly avaialble Wisconsin Breast Cancer database, created by Dr. William H. Wolberg, a physician at the University of Wisconsin Hospital at Madison, Wisconsin, USA. This dataset contains 569 samples of malignant and benign tumor cells. Dr. Wolberg used fluid samples, taken from patients with solid breast masses and an easy-to-use graphical computer program called Xcyt, which is capable of performing the analysis of cytological features based on a digital scan. The program uses a curve-fitting algorithm to compute ten features from each one of the cells in the sample and then calculates the mean value, extreme value and standard error of each feature for the image.

The first two columns in the dataset store the unique ID numbers of the samples and the corresponding diagnoses (M = malignant, B = benign), respectively. Columns 3-32 contain 30 real-valued features that have been computed from digitized images of the cell nuclei, which can be used to build a model to predict whether a tumor is benign or malignant.

The Breast Cancer Wisconsin dataset resides in the UCI Machine Learning Repository, and more information about this dataset can be found at https://archive.ics.uci.edu/ml/datasets/Breast+Cancer+Wisconsin+(Diagnostic) (https://archive.ics.uci.edu/ml/datasets/Breast+Cancer+Wisconsin+(Diagnostic)).

## Objectives

This work aims to look at visualisation options which are most helpful in treating such data.

### Supervised Learning

Supervised learning is used whenever it becomes necessary to predict a certain outcome from a given input, and examples of input/output pairs are available as labelled data. A machine learning model is built from these input/output pairs, which comprise the training set. The goal is to make accurate predictions for new, never-before-seen data. Supervised learning often requires human effort to build the training set, but afterward automates and often speeds up an otherwise laborious or infeasible task.

In [36]:

```
## Load the dataset
from IPython.display import Image
import numpy as np
import pandas as pd
from sklearn.datasets import load_breast_cancer
dataset = load_breast_cancer()
print("dataset.keys():\n", dataset.keys())
```

```
dataset.keys():
 dict_keys(['data', 'target', 'target_names', 'DESCR', 'feature_names', 'fil
ename'])
```

In [38]:

```
print("Sample counts per class:\n",
      {n: v for n, v in zip(dataset.target_names, np.bincount(dataset.target))})
```

```
Sample counts per class:
 {'malignant': 212, 'benign': 357}
```

From the above, it can be concluded that out of the 569 persons, 357 are labeled as B (benign) and 212 as M (malignant).

In [39]:

```
print("Feature names:\n", dataset.feature_names)
```

```
Feature names:
 ['mean radius' 'mean texture' 'mean perimeter' 'mean area'
 'mean smoothness' 'mean compactness' 'mean concavity'
 'mean concave points' 'mean symmetry' 'mean fractal dimension'
 'radius error' 'texture error' 'perimeter error' 'area error'
 'smoothness error' 'compactness error' 'concavity error'
 'concave points error' 'symmetry error' 'fractal dimension error'
 'worst radius' 'worst texture' 'worst perimeter' 'worst area'
 'worst smoothness' 'worst compactness' 'worst concavity'
 'worst concave points' 'worst symmetry' 'worst fractal dimension']
```

In order to investigate association of breast cancer to various medical factors, the "breast-cancer-wisconsin.csv" file was downloaded

In [63]:

```
df = pd.read_csv(r'C:\Knowledgebase\Certs\DataScience\breast-cancer-wisconsin.csv')
```

Data clean up

In [64]:

```
df.head()
```

Out[64]:

| | id | clump_thickness | size_uniformity | shape_uniformity | marginal_adhesion | epithelial_si |
|---|---|---|---|---|---|---|
| **0** | 1000025 | 5 | 1 | 1 | 1 | |
| **1** | 1002945 | 5 | 4 | 4 | 5 | |
| **2** | 1015425 | 3 | 1 | 1 | 1 | |
| **3** | 1016277 | 6 | 8 | 8 | 1 | |
| **4** | 1017023 | 4 | 1 | 1 | 3 | |

In [66]:

```
## Handling NA

df = df.drop(['bare_nucleoli'], axis=1)
df.isnull().sum()
```

Out[66]:

```
id                   0
clump_thickness      0
size_uniformity      0
shape_uniformity     0
marginal_adhesion    0
epithelial_size      0
bland_chromatin      0
normal_nucleoli      0
mitoses              0
class                0
dtype: int64
```

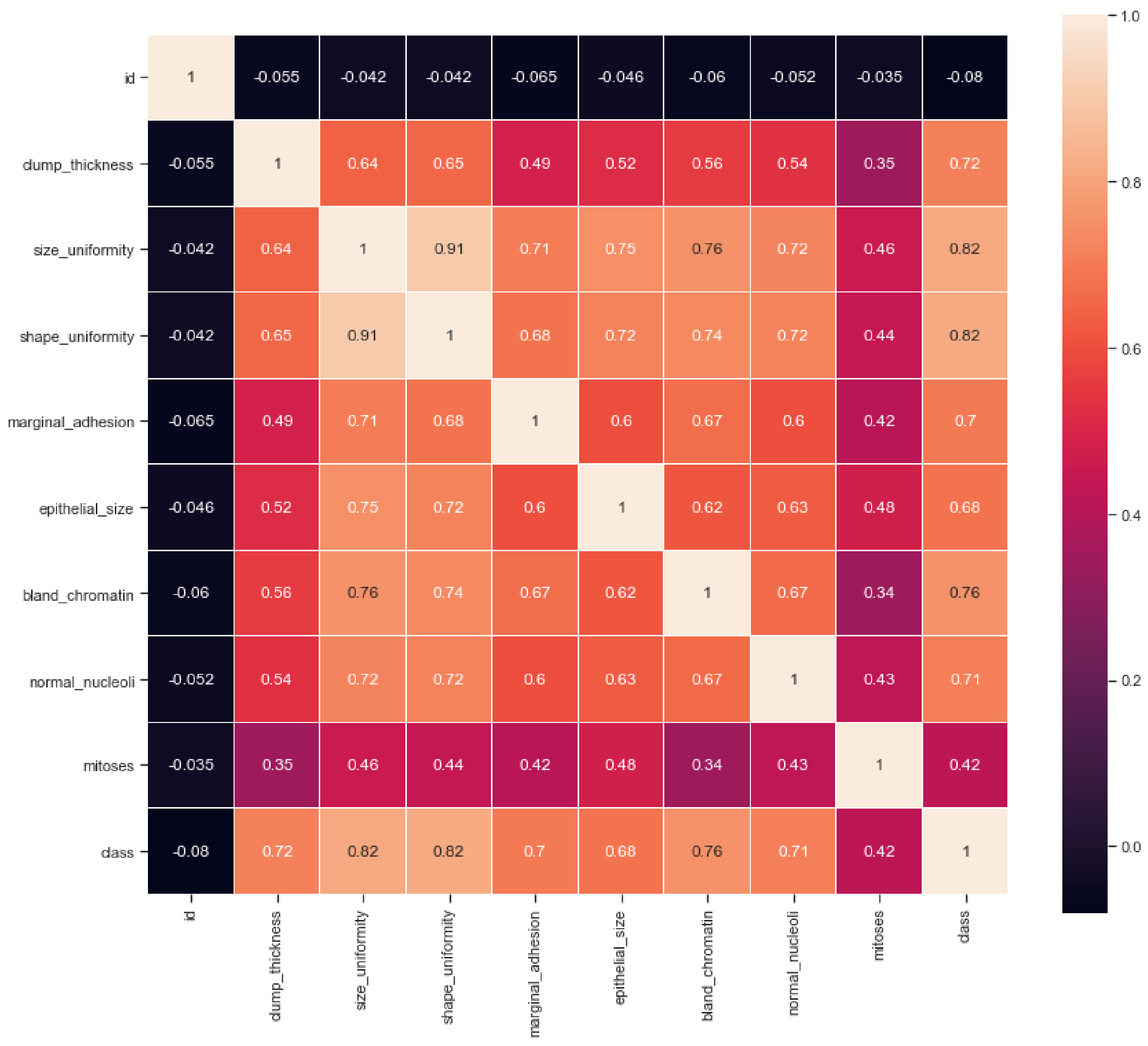
In [67]:

```
for label in ['clump_thickness','size_uniformity','shape_uniformity','marginal_adhesion', '
    df[label] = df[label].fillna(method='ffill')
```

## Heat Map

In [68]:

```
import seaborn as sns
import matplotlib.pyplot as plt
sns.set(style='ticks', color_codes=True)
plt.figure(figsize=(14, 12))
sns.heatmap(df.astype(float).corr(), linewidths=0.1, square=True, linecolor='white', annot=
plt.show()
```



| | id | clump_thickness | size_uniformity | shape_uniformity | marginal_adhesion | epithelial_size | bland_chromatin | normal_nucleoli | mitoses | class |
|---|---|---|---|---|---|---|---|---|---|---|
| id | 1 | -0.055 | -0.042 | -0.042 | -0.065 | -0.046 | -0.06 | -0.052 | -0.035 | -0.08 |
| clump_thickness | -0.055 | 1 | 0.64 | 0.65 | 0.49 | 0.52 | 0.56 | 0.54 | 0.35 | 0.72 |
| size_uniformity | -0.042 | 0.64 | 1 | 0.91 | 0.71 | 0.75 | 0.76 | 0.72 | 0.46 | 0.82 |
| shape_uniformity | -0.042 | 0.65 | 0.91 | 1 | 0.68 | 0.72 | 0.74 | 0.72 | 0.44 | 0.82 |
| marginal_adhesion | -0.065 | 0.49 | 0.71 | 0.68 | 1 | 0.6 | 0.67 | 0.6 | 0.42 | 0.7 |
| epithelial_size | -0.046 | 0.52 | 0.75 | 0.72 | 0.6 | 1 | 0.62 | 0.63 | 0.48 | 0.68 |
| bland_chromatin | -0.06 | 0.56 | 0.76 | 0.74 | 0.67 | 0.62 | 1 | 0.67 | 0.34 | 0.76 |
| normal_nucleoli | -0.052 | 0.54 | 0.72 | 0.72 | 0.6 | 0.63 | 0.67 | 1 | 0.43 | 0.71 |
| mitoses | -0.035 | 0.35 | 0.46 | 0.44 | 0.42 | 0.48 | 0.34 | 0.43 | 1 | 0.42 |
| class | -0.08 | 0.72 | 0.82 | 0.82 | 0.7 | 0.68 | 0.76 | 0.71 | 0.42 | 1 |

What we need to understand here is the co-relation among every attributes, where +1 shows the highest positive co-relativity and -1 being the negative co-relativity.

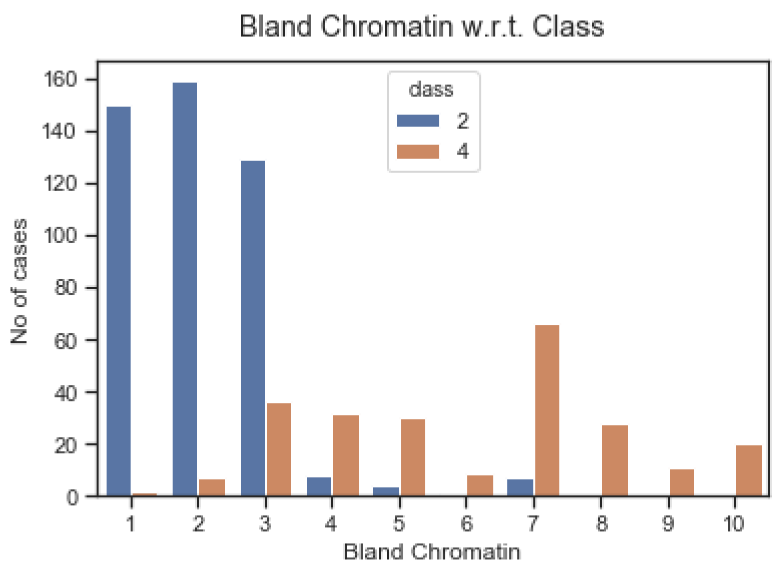
Let’s focus on the square where attribute size_uniformity of X-axis and shape_uniformity of Y -axis meet that is 0.91, which shows that these two attributes are highly co-related to each other. In more simple words, the value of size_uniformity increases when the value of shape_uniformity increases,had it been -0.91 again they are highly co-related but this time one increases when another decreases.

In [70]:

```
fig = plt.figure()
ax = sns.countplot(x='bland_chromatin', hue='class', data=df)
ax.set(xlabel='Bland Chromatin', ylabel='No of cases')
fig.suptitle('Bland Chromatin w.r.t. Class', y=0.96)
```

Out[70]:

```
Text(0.5, 0.96, 'Bland Chromatin w.r.t. Class')
```



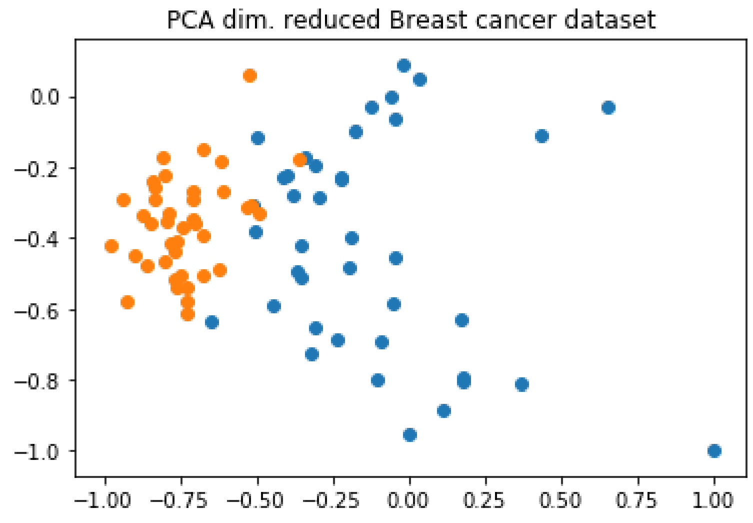
In [ ]:

In [ ]:

```
import numpy as np
import scipy
from scipy.linalg import expm
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, MinMaxScaler
from sklearn.decomposition import PCA

def Breast_cancer(training_size, test_size, n, PLOT_DATA):
    class_labels = [r'A', r'B']
    data, target = datasets.load_breast_cancer(True)
    sample_train, sample_test, label_train, label_test = train_test_split(data, target, tes

    # Now we standarize for gaussian around 0 with unit variance
    std_scale = StandardScaler().fit(sample_train)
    sample_train = std_scale.transform(sample_train)
    sample_test = std_scale.transform(sample_test)

    # Now reduce number of features to number of qubits
    pca = PCA(n_components=n).fit(sample_train)
    sample_train = pca.transform(sample_train)
    sample_test = pca.transform(sample_test)

    # Scale to the range (-1,+1)
    samples = np.append(sample_train, sample_test, axis=0)
    minmax_scale = MinMaxScaler((-1, 1)).fit(samples)
    sample_train = minmax_scale.transform(sample_train)
    sample_test = minmax_scale.transform(sample_test)

    # Pick training size number of samples from each distro
    training_input = {key: (sample_train[label_train == k, :])[:training_size] for k, key i
    test_input = {key: (sample_train[label_train == k, :])[training_size:(
        training_size+test_size)] for k, key in enumerate(class_labels)}

    if PLOT_DATA:
        for k in range(0, 2):
            plt.scatter(sample_train[label_train == k, 0][:training_size],
                        sample_train[label_train == k, 1][:training_size])

        plt.title("PCA dim. reduced Breast cancer dataset")
        plt.show()

    return sample_train, training_input, test_input, class_labels
```

In [10]:

```
from qiskit.aqua.utils import split_dataset_to_data_and_labels
from qiskit.aqua.input import energy_input
from qiskit.aqua import run_algorithm
import numpy as np

n = 2  # dimension of each data point
sample_Total, training_input, test_input, class_labels = Breast_cancer(training_size=40,
                                                              test_size=10, n=n, PLOT_DATA=

temp = [test_input[k] for k in test_input]
total_array = np.concatenate(temp)

aqua_dict = {
    'problem': {'name': 'svm_classification', 'random_seed': 100},
    'algorithm': {
        'name': 'QSVM.Kernel'
    },
    'feature_map': {'name': 'SecondOrderExpansion', 'depth': 2, 'entangler_map': {0: [1]}},
    'multiclass_extension': {'name': 'AllPairs'},
    'backend': {'name': 'qasm_simulator', 'shots': 256}
}
```



In [ ]: