**Using Predictive Analysis To Predict Diagnosis of a Breast Tumor**

**Part\_1: Problem Statement**

Breast cancer is the most common malignancy among women, accounting for nearly 1 in 3 cancers diagnosed among women in the United States, and it is the second leading cause of cancer death among women. Breast Cancer occurs as a result of abnormal growth of cells in the breast tissue, commonly referred to as a Tumor. A tumor does not mean cancer - tumors can be benign (not cancerous), pre-malignant (pre-cancerous), or malignant (cancerous). Tests such as MRI, mammogram, ultrasound, and biopsy are commonly used to diagnose breast cancer performed.

**1.1 Expected outcome**

Given breast cancer results from breast fine-needle aspiration (FNA) test (is a quick and simple procedure to perform, which removes some fluid or cells from a breast lesion or cyst (a lump, sore, or swelling) with a fine needle similar to a blood sample needle). Since this build a model that can classify a breast cancer tumor using two training classification:

* 1 = Malignant (Cancerous) - Present
* 0 = Benign (Not Cancerous) -Absent

**1.2 Objective**

Since the labels in the data are discrete, the predication falls into two categories, (i.e. Malignant or benign). In machine learning, this is a classification problem.

*Thus, the goal is to classify whether the breast cancer is benign or malignant and predict the recurrence and non-recurrence of malignant cases after a certain period. To achieve this we have used machine learning classification methods to fit a function that can predict the discrete class of new input.*

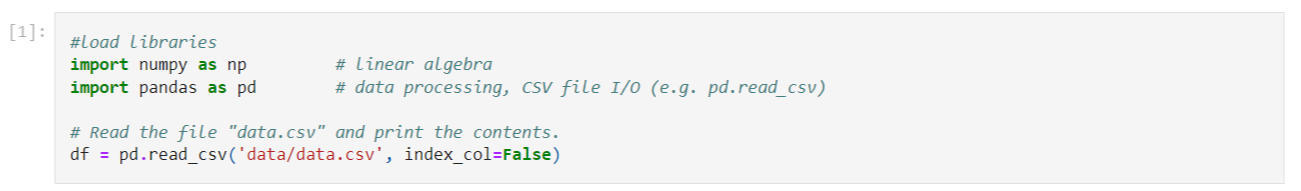
**1.3 Identify data sources**

The [**Breast Cancer**](https://archive.ics.uci.edu/ml/datasets/Breast+Cancer+Wisconsin+%28Diagnostic%29) datasets is available as machine learning repository maintained by the University of California, Irvine. The dataset contains **569 samples of malignant and benign tumor cells**.

* The first two columns in the dataset store the unique ID numbers of the samples and the corresponding diagnosis (M = malignant, B = benign), respectively.
* The columns 3-32 contain 30 real-value 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.

**Getting Started: Load libraries and set options**

In [1]:

*#load libraries*

**Load Dataset**

First, load the supplied CSV file using additional options in the Pandas **read\_csv** function.

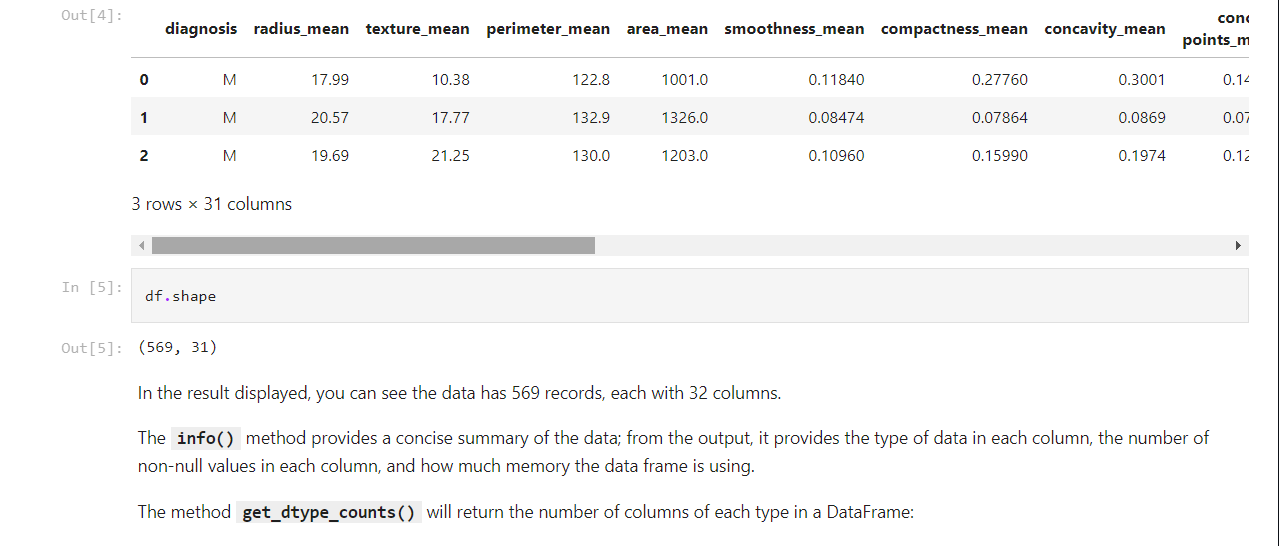
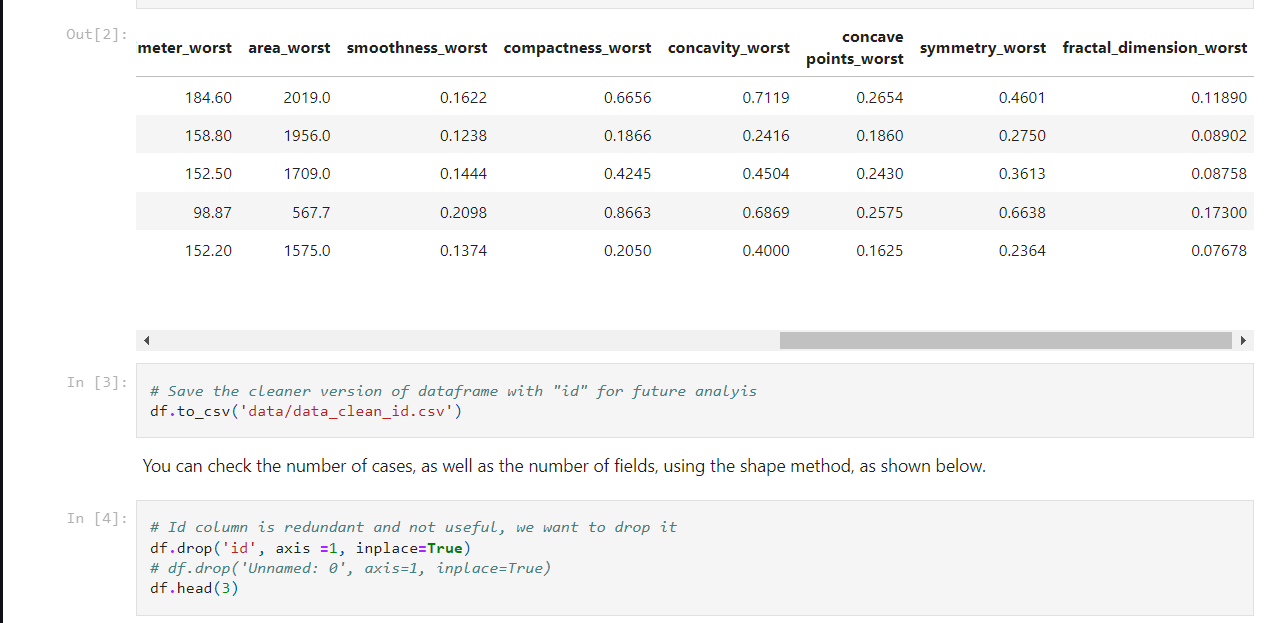
**Inspecting the data**

The first step is to visually inspect the new data set. There are multiple ways to achieve this:

* The easiest being to request the first few records using the DataFrame **data.head()** method. By default, **data.head()** returns the first 5 rows from the DataFrame object df (excluding the header row).
* Alternatively, one can also use **df.tail()** to return the five rows of the data frame.
* For both head and tail methods, there is an option to specify the number of records by including the required number in between the parentheses when calling either method.Inspecting the data

In [2]:

df**.**head()

Out[2]:

5 rows × 32 columns

In [3]:

*# Save the cleaner version of dataframe with "id" for future analyis*

df**.**to\_csv('data/data\_clean\_id.csv')

You can check the number of cases, as well as the number of fields, using the shape method, as shown below.

In [4]:

*# Id column is redundant and not useful, we want to drop it*

df**.**drop('id', axis **=**1, inplace**=True**)

*# df.drop('Unnamed: 0', axis=1, inplace=True)*

df**.**head(3)

Out[4]:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| **2** | M |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | 0.2430 | 0.3613 | 0.08758 |

3 rows × 31 columns

In [5]:

df**.**shape

Out[5]:

(569, 31)

In the result displayed, you can see the data has 569 records, each with 32 columns.

The **info()** method provides a concise summary of the data; from the output, it provides the type of data in each column, the number of non-null values in each column, and how much memory the data frame is using.

The method **get\_dtype\_counts()** will return the number of columns of each type in a DataFrame:

In [6]:

*# Review data types with "info()".*

df**.**info()

<class 'pandas.core.frame.DataFrame'>

RangeIndex: 569 entries, 0 to 568

Data columns (total 31 columns):

# Column Non-Null Count Dtype

--- ------ -------------- -----

0 diagnosis 569 non-null object

1 radius\_mean 569 non-null float64

2 texture\_mean 569 non-null float64

3 perimeter\_mean 569 non-null float64

4 area\_mean 569 non-null float64

5 smoothness\_mean 569 non-null float64

6 compactness\_mean 569 non-null float64

7 concavity\_mean 569 non-null float64

8 concave points\_mean 569 non-null float64

9 symmetry\_mean 569 non-null float64

10 fractal\_dimension\_mean 569 non-null float64

11 radius\_se 569 non-null float64

12 texture\_se 569 non-null float64

13 perimeter\_se 569 non-null float64

14 area\_se 569 non-null float64

15 smoothness\_se 569 non-null float64

16 compactness\_se 569 non-null float64

17 concavity\_se 569 non-null float64

18 concave points\_se 569 non-null float64

19 symmetry\_se 569 non-null float64

20 fractal\_dimension\_se 569 non-null float64

21 radius\_worst 569 non-null float64

22 texture\_worst 569 non-null float64

23 perimeter\_worst 569 non-null float64

24 area\_worst 569 non-null float64

25 smoothness\_worst 569 non-null float64

26 compactness\_worst 569 non-null float64

27 concavity\_worst 569 non-null float64

28 concave points\_worst 569 non-null float64

29 symmetry\_worst 569 non-null float64

30 fractal\_dimension\_worst 569 non-null float64

dtypes: float64(30), object(1)

memory usage: 137.9+ KB

In [7]:

*# Check for missing variables*

df**.**isnull()**.**any()

Out[7]:

diagnosis False

radius\_mean False

texture\_mean False

perimeter\_mean False

area\_mean False

smoothness\_mean False

compactness\_mean False

concavity\_mean False

concave points\_mean False

symmetry\_mean False

fractal\_dimension\_mean False

radius\_se False

texture\_se False

perimeter\_se False

area\_se False

smoothness\_se False

compactness\_se False

concavity\_se False

concave points\_se False

symmetry\_se False

fractal\_dimension\_se False

radius\_worst False

texture\_worst False

perimeter\_worst False

area\_worst False

smoothness\_worst False

compactness\_worst False

concavity\_worst False

concave points\_worst False

symmetry\_worst False

fractal\_dimension\_worst False

dtype: bool

In [8]:

df**.**diagnosis**.**unique()

Out[8]:

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

From the results above, diagnosis is a categorical variable, because it represents a fix number of possible values (i.e, Malignant, of Benign. The machine learning algorithms wants numbers, and not strings, as their inputs so we need some method of coding to convert them.

In [9]:

*# Save the cleaner version of dataframe for future analyis*

df**.**to\_csv('data/data\_clean.csv')

**NOTE:** Now that we have a good intuitive sense of the data, Next step involves taking a closer look at attributes and data values. In Part\_2, we will explore the data further.

**Part 2: Exploratory Data Analysis**

Now that we have a good intuitive sense of the data, Next step involves taking a closer look at attributes and data values. In this section, I am getting familiar with the data, which will provide useful knowledge for data pre-processing.

**2.1 Objectives of Data Exploration**

Exploratory data analysis (EDA) is a very important step which takes place after feature engineering and acquiring data and it should be done before any modeling. This is because it is very important for a data scientist to be able to understand the nature of the data without making assumptions. The results of data exploration can be extremely useful in grasping the structure of the data, the distribution of the values, and the presence of extreme values and interrelationships within the data set.

**The purpose of EDA is:**

* to use summary statistics and visualizations to better understand data,

\*find clues about the tendencies of the data, its quality and to formulate assumptions and the hypothesis of our analysis

* For data preprocessing to be successful, it is essential to have an overall picture of your data

Basic statistical descriptions can be used to identify properties of the data and highlight which data values should be treated as noise or outliers.\*\*

Next step is to explore the data. There are two approached used to examine the data using:

1. \***Descriptive statistics**\* is the process of condensing key characteristics of the data set into simple numeric metrics. Some of the common metrics used are mean, standard deviation, and correlation.
2. \***Visualization**\* is the process of projecting the data, or parts of it, into Cartesian space or into abstract images. In the data mining process, data exploration is leveraged in many different steps including preprocessing, modeling, and interpretation of results.

**2.2 Descriptive statistics**

Summary statistics are measurements meant to describe data. In the field of descriptive statistics, there are many [**summary measurements**](http://www.saedsayad.com/numerical_variables.htm).

In [10]:

**%matplotlib** inline

**import** matplotlib.pyplot **as** plt

*#Load libraries for data processing*

**import** pandas **as** pd *#data processing, CSV file I/O (e.g. pd.read\_csv)*

**import** numpy **as** np

**from** scipy.stats **import** norm

**import** seaborn **as** sns *# data visualization*

plt**.**rcParams['figure.figsize'] **=** (15,8)

plt**.**rcParams['axes.titlesize'] **=** 'large'

In [11]:

*# usind clean data with "id"*

df **=** pd**.**read\_csv('data/data\_clean\_id.csv', index\_col**=False**)

df**.**drop('Unnamed: 0',axis**=**1, inplace**=True**)

df**.**head(3)

Out[11]:

|  | **id** | **diagnosis** | **radius\_mean** | **texture\_mean** | **perimeter\_mean** | **area\_mean** | **smoothness\_mean** | **compactness\_mean** | **concavity\_mean** | **concave points\_mean** | **...** | **radius\_worst** | **texture\_worst** | **perimeter\_worst** | **area\_worst** | **smoothness\_worst** | **compactness\_worst** | **concavity\_worst** | **concave points\_worst** | **symmetry\_worst** | **fractal\_dimension\_worst** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | 842302 | M | 17.99 | 10.38 | 122.8 | 1001.0 | 0.11840 | 0.27760 | 0.3001 | 0.14710 | ... | 25.38 | 17.33 | 184.6 | 2019.0 | 0.1622 | 0.6656 | 0.7119 | 0.2654 | 0.4601 | 0.11890 |
| **1** | 842517 | M | 20.57 | 17.77 | 132.9 | 1326.0 | 0.08474 | 0.07864 | 0.0869 | 0.07017 | ... | 24.99 | 23.41 | 158.8 | 1956.0 | 0.1238 | 0.1866 | 0.2416 | 0.1860 | 0.2750 | 0.08902 |
| **2** | 84300903 | M | 19.69 | 21.25 | 130.0 | 1203.0 | 0.10960 | 0.15990 | 0.1974 | 0.12790 | ... | 23.57 | 25.53 | 152.5 | 1709.0 | 0.1444 | 0.4245 | 0.4504 | 0.2430 | 0.3613 | 0.08758 |

3 rows × 32 columns

In [12]:

*#basic descriptive statistics*

df**.**describe()

Out[12]:

|  | **id** | **radius\_mean** | **texture\_mean** | **perimeter\_mean** | **area\_mean** | **smoothness\_mean** | **compactness\_mean** | **concavity\_mean** | **concave points\_mean** | **symmetry\_mean** | **...** | **radius\_worst** | **texture\_worst** | **perimeter\_worst** | **area\_worst** | **smoothness\_worst** | **compactness\_worst** | **concavity\_worst** | **concave points\_worst** | **symmetry\_worst** | **fractal\_dimension\_worst** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **count** | 5.690000e+02 | 569.000000 | 569.000000 | 569.000000 | 569.000000 | 569.000000 | 569.000000 | 569.000000 | 569.000000 | 569.000000 | ... | 569.000000 | 569.000000 | 569.000000 | 569.000000 | 569.000000 | 569.000000 | 569.000000 | 569.000000 | 569.000000 | 569.000000 |
| **mean** | 3.037183e+07 | 14.127292 | 19.289649 | 91.969033 | 654.889104 | 0.096360 | 0.104341 | 0.088799 | 0.048919 | 0.181162 | ... | 16.269190 | 25.677223 | 107.261213 | 880.583128 | 0.132369 | 0.254265 | 0.272188 | 0.114606 | 0.290076 | 0.083946 |
| **std** | 1.250206e+08 | 3.524049 | 4.301036 | 24.298981 | 351.914129 | 0.014064 | 0.052813 | 0.079720 | 0.038803 | 0.027414 | ... | 4.833242 | 6.146258 | 33.602542 | 569.356993 | 0.022832 | 0.157336 | 0.208624 | 0.065732 | 0.061867 | 0.018061 |
| **min** | 8.670000e+03 | 6.981000 | 9.710000 | 43.790000 | 143.500000 | 0.052630 | 0.019380 | 0.000000 | 0.000000 | 0.106000 | ... | 7.930000 | 12.020000 | 50.410000 | 185.200000 | 0.071170 | 0.027290 | 0.000000 | 0.000000 | 0.156500 | 0.055040 |
| **25%** | 8.692180e+05 | 11.700000 | 16.170000 | 75.170000 | 420.300000 | 0.086370 | 0.064920 | 0.029560 | 0.020310 | 0.161900 | ... | 13.010000 | 21.080000 | 84.110000 | 515.300000 | 0.116600 | 0.147200 | 0.114500 | 0.064930 | 0.250400 | 0.071460 |
| **50%** | 9.060240e+05 | 13.370000 | 18.840000 | 86.240000 | 551.100000 | 0.095870 | 0.092630 | 0.061540 | 0.033500 | 0.179200 | ... | 14.970000 | 25.410000 | 97.660000 | 686.500000 | 0.131300 | 0.211900 | 0.226700 | 0.099930 | 0.282200 | 0.080040 |
| **75%** | 8.813129e+06 | 15.780000 | 21.800000 | 104.100000 | 782.700000 | 0.105300 | 0.130400 | 0.130700 | 0.074000 | 0.195700 | ... | 18.790000 | 29.720000 | 125.400000 | 1084.000000 | 0.146000 | 0.339100 | 0.382900 | 0.161400 | 0.317900 | 0.092080 |
| **max** | 9.113205e+08 | 28.110000 | 39.280000 | 188.500000 | 2501.000000 | 0.163400 | 0.345400 | 0.426800 | 0.201200 | 0.304000 | ... | 36.040000 | 49.540000 | 251.200000 | 4254.000000 | 0.222600 | 1.058000 | 1.252000 | 0.291000 | 0.663800 | 0.207500 |

8 rows × 31 columns

In [13]:

df**.**skew()

Out[13]:

id 6.473752

radius\_mean 0.942380

texture\_mean 0.650450

perimeter\_mean 0.990650

area\_mean 1.645732

smoothness\_mean 0.456324

compactness\_mean 1.190123

concavity\_mean 1.401180

concave points\_mean 1.171180

symmetry\_mean 0.725609

fractal\_dimension\_mean 1.304489

radius\_se 3.088612

texture\_se 1.646444

perimeter\_se 3.443615

area\_se 5.447186

smoothness\_se 2.314450

compactness\_se 1.902221

concavity\_se 5.110463

concave points\_se 1.444678

symmetry\_se 2.195133

fractal\_dimension\_se 3.923969

radius\_worst 1.103115

texture\_worst 0.498321

perimeter\_worst 1.128164

area\_worst 1.859373

smoothness\_worst 0.415426

compactness\_worst 1.473555

concavity\_worst 1.150237

concave points\_worst 0.492616

symmetry\_worst 1.433928

fractal\_dimension\_worst 1.662579

dtype: float64

The skew result show a positive (right) or negative (left) skew. Values closer to zero show less skew.

From the graphs, we can see that **radius\_mean**, **perimeter\_mean**, **area\_mean**, **concavity\_mean** and **concave\_points\_mean** are useful in predicting cancer type due to the distinct grouping between malignant and benign cancer types in these features. We can also see that **area\_worst** and **perimeter\_worst** are also quite useful.

In [14]:

df**.**diagnosis**.**unique()

Out[14]:

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

In [15]:

*# Group by diagnosis and review the output.*

diag\_gr **=** df**.**groupby('diagnosis', axis**=**0)

pd**.**DataFrame(diag\_gr**.**size(), columns**=**['# of observations'])

Out[15]:

|  | **# of observations** |
| --- | --- |
| **diagnosis** |  |
| **B** | 357 |
| **M** | 212 |

Check binary encoding from NB1 to confirm the coversion of the diagnosis categorical data into numeric, where

* Malignant = 1 (indicates prescence of cancer cells)
* Benign = 0 (indicates abscence)

**Observation**

*357 observations indicating the absence of cancer cells and 212 show absence of cancer cell*

Lets confirm this, by ploting the histogram

**2.3 Unimodal Data Visualizations**

One of the main goals of visualizing the data here is to observe which features are most helpful in predicting malignant or benign cancer. The other is to see general trends that may aid us in model selection and hyper parameter selection.

Apply 3 techniques that you can use to understand each attribute of your dataset independently.

* Histograms.
* Density Plots.
* Box and Whisker Plots.

In [16]:

*#lets get the frequency of cancer diagnosis*

sns**.**set\_style("white")

sns**.**set\_context({"figure.figsize": (10, 8)})

sns**.**countplot(df['diagnosis'],label**=**'Count',palette**=**"Set3")

C:\ProgramData\Anaconda3\lib\site-packages\seaborn\\_decorators.py:36: FutureWarning: Pass the following variable as a keyword arg: x. From version 0.12, the only valid positional argument will be `data`, and passing other arguments without an explicit keyword will result in an error or misinterpretation.

warnings.warn(

Out[16]:

<AxesSubplot:xlabel='diagnosis', ylabel='count'>

**2.3.1 Visualise distribution of data via histograms**

Histograms are commonly used to visualize numerical variables. A histogram is similar to a bar graph after the values of the variable are grouped (binned) into a finite number of intervals (bins).

Histograms group data into bins and provide you a count of the number of observations in each bin. From the shape of the bins you can quickly get a feeling for whether an attribute is Gaussian, skewed or even has an exponential distribution. It can also help you see possible outliers.

**Separate columns into smaller dataframes to perform visualization**

In [17]:

*#Break up columns into groups, according to their suffix designation*

*#(\_mean, \_se,and \_\_worst) to perform visualisation plots off.*

*#Join the 'ID' and 'Diagnosis' back on*

df\_id\_diag**=**df**.**loc[:,["id","diagnosis"]]

df\_diag**=**df**.**loc[:,["diagnosis"]]

*#For a merge + slice:*

df\_mean**=**df**.**iloc[:,1:11]

df\_se**=**df**.**iloc[:,11:22]

df\_worst**=**df**.**iloc[:,23:]

print(df\_id\_diag**.**columns)

*#print(data\_mean.columns)*

*#print(data\_se.columns)*

*#print(data\_worst.columns)*

Index(['id', 'diagnosis'], dtype='object')

**Histogram the \_mean suffix designition**

In [18]:

*#Plot histograms of CUT1 variables*

hist\_mean**=**df\_mean**.**hist(bins**=**10, figsize**=**(15, 10),grid**=False**,)

*#Any individual histograms, use this:*

*#df\_cut['radius\_worst'].hist(bins=100)*

**Histogram for the \_se suffix designition**

In [19]:

*#Plot histograms of \_se variables*

hist\_se**=**df\_se**.**hist(bins**=**10, figsize**=**(15, 10),grid**=False**,)

**Histogram \_worst suffix designition**

In [20]:

*#Plot histograms of \_worst variables*

hist\_worst**=**df\_worst**.**hist(bins**=**10, figsize**=**(15, 10),grid**=False**,)

**Observation**

We can see that perhaps the attributes **concavity**,and **concavity\_point**may have an exponential distribution ( ). We can also see that perhaps the texture and smooth and symmetry attributes may have a Gaussian or nearly Gaussian distribution. This is interesting because many machine learning techniques assume a Gaussian univariate distribution on the input variables.

**2.3.2 Visualize distribution of data via density plots**

**Density plots \_mean suffix designition**

In [21]:

*#Density Plots*

plt **=** df\_mean**.**plot(kind**=** 'density', subplots**=True**, layout**=**(4,3), sharex**=False**,

sharey**=False**, fontsize**=**12, figsize**=**(15,10))

**Density plots \_se suffix designition**

In [22]:

*#Density Plots*

plt **=** df\_se**.**plot(kind**=** 'density', subplots**=True**, layout**=**(4,3), sharex**=False**,

sharey**=False**, fontsize**=**12, figsize**=**(15,10))

**Density plot \_worst suffix designition**

In [23]:

*#Density Plots*

plt **=** df\_worst**.**plot(kind**=** 'kde', subplots**=True**, layout**=**(4,3), sharex**=False**, sharey**=False**,

fontsize**=**5, figsize**=**(15,10))

**Observation**

We can see that perhaps the attributes perimeter,radius, area, concavity, compactness may have an exponential distribution( ). We can also see that perhaps the texture and smooth and symmetry attributes may have a Gaussian or nearly Gaussian distribution. This is interesting because many machine learning techniques assume a Gaussian univariate distribution on the input variables.

**2.3.3 Visualise distribution of data via box plots**

**Box plot \_mean suffix designition**

In [24]:

*# box and whisker plots*

plt**=**df\_mean**.**plot(kind**=** 'box' , subplots**=True**, layout**=**(4,4), sharex**=False**, sharey**=False**,

fontsize**=**12)

**Box plot \_se suffix designition**

In [25]:

*# box and whisker plots*

plt**=**df\_se**.**plot(kind**=** 'box' , subplots**=True**, layout**=**(4,4), sharex**=False**, sharey**=False**,

fontsize**=**12)

**Box plot \_worst suffix designition**

In [26]:

*# box and whisker plots*

plt**=**df\_worst**.**plot(kind**=** 'box' , subplots**=True**, layout**=**(4,4), sharex**=False**, sharey**=False**,

fontsize**=**12)

**Observation**

We can see that perhaps the attributes perimeter,radius, area, concavity,ompactness may have an exponential distribution( ). We can also see that perhaps the texture and smooth and symmetry attributes may have a Gaussian or nearly Gaussian distribution. This is interesting because many machine learning techniques assume a Gaussian univariate distribution on the input variables.

**2.4 Multimodal Data Visualizations**

* Scatter plots
* Correlation matrix

**Correlation matrix**

In [27]:

*# plot correlation matrix*

**import** pandas **as** pd

**import** numpy **as** np

**import** seaborn **as** sns

**from** matplotlib **import** pyplot **as** plt

plt**.**style**.**use('fivethirtyeight')

sns**.**set\_style("white")

df **=** pd**.**read\_csv('data/data\_clean.csv', index\_col**=False**)

df**.**drop('Unnamed: 0',axis**=**1, inplace**=True**)

*# Compute the correlation matrix*

corr **=** df\_mean**.**corr()

*# Generate a mask for the upper triangle*

mask **=** np**.**zeros\_like(corr, dtype**=**np**.**bool)

mask[np**.**triu\_indices\_from(mask)] **=** **True**

*# Set up the matplotlib figure*

df, ax **=** plt**.**subplots(figsize**=**(8, 8))

plt**.**title('Breast Cancer Feature Correlation')

*# Generate a custom diverging colormap*

cmap **=** sns**.**diverging\_palette(260, 10, as\_cmap**=True**)

*# Draw the heatmap with the mask and correct aspect ratio*

sns**.**heatmap(corr, vmax**=**1.2, square**=**'square', cmap**=**cmap, mask**=**mask,

ax**=**ax,annot**=True**, fmt**=**'.2g',linewidths**=**2)

<ipython-input-27-13301591d600>:17: DeprecationWarning: `np.bool` is a deprecated alias for the builtin `bool`. To silence this warning, use `bool` by itself. Doing this will not modify any behavior and is safe. If you specifically wanted the numpy scalar type, use `np.bool\_` here.

Deprecated in NumPy 1.20; for more details and guidance: https://numpy.org/devdocs/release/1.20.0-notes.html#deprecations

mask = np.zeros\_like(corr, dtype=np.bool)

Out[27]:

<AxesSubplot:title={'center':'Breast Cancer Feature Correlation'}>

**Observation:**

We can see strong positive relationship exists with mean values paramaters between **1** to **0.75**.

* The mean area of the tissue nucleus has a strong positive correlation with mean values of radius and parameter;
* Some paramters are moderately positive corrlated (r between 0.5-0.75)are concavity and area, concavity and perimeter etc
* Likewise, we see some strong negative correlation between fractal\_dimension with radius, texture, parameter mean values.

In [28]:

plt**.**style**.**use('fivethirtyeight')

sns**.**set\_style("white")

df **=** pd**.**read\_csv('data/data\_clean.csv', index\_col**=False**)

g **=** sns**.**PairGrid(df[[df**.**columns[1],df**.**columns[2], df**.**columns[3],

df**.**columns[4], df**.**columns[5], df**.**columns[6]]], hue**=**'diagnosis')

g **=** g**.**map\_diag(plt**.**hist)

g **=** g**.**map\_offdiag(plt**.**scatter, s **=** 3)

**Part\_2 Summary:**

* Mean values of cell radius, perimeter, area, compactness, concavity and concave points can be used in classification of the cancer. Larger values of these parameters tends to show a correlation with malignant tumors.
* mean values of texture, smoothness, symmetry or fractual dimension does not show a particular preference of one diagnosis over the other.
* In any of the histograms there are no noticeable large outliers that warrants further cleanup.

**Part\_3: Pre-Processing the data**

**Introduction**

[Data preprocessing](http://www.cs.ccsu.edu/~markov/ccsu_courses/datamining-3.html) is a crucial step for any data analysis problem. It is often a very good idea to prepare your data in such way to best expose the structure of the problem to the machine learning algorithms that you intend to use.This involves a number of activities such as:

* Assigning numerical values to categorical data;
* Handling missing values; and
* Normalizing the features (so that features on small scales do not dominate when fitting a model to the data).

In Part\_2, I explored the data, to help gain insight on the distribution of the data as well as how the attributes correlate to each other. I identified some features of interest. In this notebook I use feature selection to reduce high-dimension data, feature extraction and transformation for dimensionality reduction.

**Goal:**

Find the most predictive features of the data and filter it so it will enhance the predictive power of the analytics model.

**Load data and essential libraries**

In [29]:

**%matplotlib** inline

**import** matplotlib.pyplot **as** plt

*#Load libraries for data processing*

**import** pandas **as** pd *#data processing, CSV file I/O (e.g. pd.read\_csv)*

**import** numpy **as** np

**from** scipy.stats **import** norm

*# visualization*

**import** seaborn **as** sns

plt**.**style**.**use('fivethirtyeight')

sns**.**set\_style("white")

plt**.**rcParams['figure.figsize'] **=** (8,4)

*#plt.rcParams['axes.titlesize'] = 'large'*

df **=** pd**.**read\_csv('data/data\_clean.csv', index\_col**=False**)

df**.**drop('Unnamed: 0',axis**=**1, inplace**=True**)

df**.**head(3)

Out[29]:

|  | **diagnosis** | **radius\_mean** | **texture\_mean** | **perimeter\_mean** | **area\_mean** | **smoothness\_mean** | **compactness\_mean** | **concavity\_mean** | **concave points\_mean** | **symmetry\_mean** | **...** | **radius\_worst** | **texture\_worst** | **perimeter\_worst** | **area\_worst** | **smoothness\_worst** | **compactness\_worst** | **concavity\_worst** | **concave points\_worst** | **symmetry\_worst** | **fractal\_dimension\_worst** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | M | 17.99 | 10.38 | 122.8 | 1001.0 | 0.11840 | 0.27760 | 0.3001 | 0.14710 | 0.2419 | ... | 25.38 | 17.33 | 184.6 | 2019.0 | 0.1622 | 0.6656 | 0.7119 | 0.2654 | 0.4601 | 0.11890 |
| **1** | M | 20.57 | 17.77 | 132.9 | 1326.0 | 0.08474 | 0.07864 | 0.0869 | 0.07017 | 0.1812 | ... | 24.99 | 23.41 | 158.8 | 1956.0 | 0.1238 | 0.1866 | 0.2416 | 0.1860 | 0.2750 | 0.08902 |
| **2** | M | 19.69 | 21.25 | 130.0 | 1203.0 | 0.10960 | 0.15990 | 0.1974 | 0.12790 | 0.2069 | ... | 23.57 | 25.53 | 152.5 | 1709.0 | 0.1444 | 0.4245 | 0.4504 | 0.2430 | 0.3613 | 0.08758 |

3 rows × 31 columns

**Label encoding**

Here, I assign the 30 features to a NumPy array X, and transform the class labels from their original string representation (M and B) into integers

In [30]:

*#Assign predictors to a variable of ndarray (matrix) type*

array **=** df**.**values

X **=** array[:,1:31]

y **=** array[:,0]

X

Out[30]:

array([[17.99, 10.38, 122.8, ..., 0.2654, 0.4601, 0.1189],

[20.57, 17.77, 132.9, ..., 0.186, 0.275, 0.08902],

[19.69, 21.25, 130.0, ..., 0.243, 0.3613, 0.08758],

...,

[16.6, 28.08, 108.3, ..., 0.1418, 0.2218, 0.0782],

[20.6, 29.33, 140.1, ..., 0.265, 0.4087, 0.124],

[7.76, 24.54, 47.92, ..., 0.0, 0.2871, 0.07039]], dtype=object)

In [31]:

*#transform the class labels from their original string representation (M and B) into integers*

**from** sklearn.preprocessing **import** LabelEncoder

le **=** LabelEncoder()

y **=** le**.**fit\_transform(y)

y

*# Call the transform method of LabelEncorder on two dummy variables*

*# le.transform (['M', 'B'])*

Out[31]:

array([1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0,

1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1,

1, 1, 0, 1, 0, 0, 0, 0, 0, 1, 1, 0, 1, 1, 0, 0, 0, 0, 1, 0, 1, 1,

0, 0, 0, 0, 1, 0, 1, 1, 0, 1, 0, 1, 1, 0, 0, 0, 1, 1, 0, 1, 1, 1,

0, 0, 0, 1, 0, 0, 1, 1, 0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 0, 0, 1, 0,

0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 1, 1, 0, 0, 0, 1, 1, 0, 1, 0, 1,

1, 0, 1, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0,

0, 0, 1, 0, 0, 0, 0, 1, 1, 0, 1, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 0,

0, 1, 0, 0, 1, 1, 1, 0, 1, 0, 1, 0, 0, 0, 1, 0, 0, 1, 1, 0, 1, 1,

1, 1, 0, 1, 1, 1, 0, 1, 0, 1, 0, 0, 1, 0, 1, 1, 1, 1, 0, 0, 1, 1,

0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 1, 0, 0, 1, 0, 0, 1, 1, 0, 1, 0, 0,

0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

1, 1, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 1, 0, 0, 1, 0, 1, 1, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1, 1,

1, 0, 0, 0, 0, 1, 0, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1,

1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 1, 1, 1, 0, 1, 1,

0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 1, 1, 0, 0,

0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 1,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 0, 1, 0, 0, 0, 0,

0, 1, 0, 0, 1, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1,

0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0,

0, 0, 0, 1, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 1, 0, 1, 0, 1, 0, 0,

0, 0, 0, 1, 0, 0, 1, 0, 1, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 0])

*After encoding the class labels(diagnosis) in an array y, the malignant tumors are now represented as class 1(i.e prescence of cancer cells) and the benign tumors are represented as class 0 (i.e no cancer cells detection), respectively*, illustrated by calling the transform method of LabelEncorder on two dummy variables.\*\*

**Assesing Model Accuracy: Split data into training and test sets**

The simplest method to evaluate the performance of a machine learning algorithm is to use different training and testing datasets. Here I will

* Split the available data into a training set and a testing set. (70% training, 30% test)
* Train the algorithm on the first part,
* make predictions on the second part and
* evaluate the predictions against the expected results.

The size of the split can depend on the size and specifics of your dataset, although it is common to use 67% of the data for training and the remaining 33% for testing.

In [32]:

**from** sklearn.model\_selection **import** train\_test\_split

*##Split data set in train 70% and test 30%*

X\_train, X\_test, y\_train, y\_test **=** train\_test\_split( X, y, test\_size**=**0.25, random\_state**=**7)

X\_train**.**shape, y\_train**.**shape, X\_test**.**shape, y\_test**.**shape

Out[32]:

((426, 30), (426,), (143, 30), (143,))

**Feature Standardization**

* Standardization is a useful technique to transform attributes with a Gaussian distribution and differing means and standard deviations to a standard Gaussian distribution with a mean of 0 and a standard deviation of 1.
* As seen in **Part\_2** the raw data has differing distributions which may have an impact on the most ML algorithms. Most machine learning and optimization algorithms behave much better if features are on the same scale.

Let’s evaluate the same algorithms with a standardized copy of the dataset. Here, I use sklearn to scale and transform the data such that each attribute has a mean value of zero and a standard deviation of one

In [33]:

**from** sklearn.preprocessing **import** StandardScaler

*# Normalize the data (center around 0 and scale to remove the variance).*

scaler **=**StandardScaler()

Xs **=** scaler**.**fit\_transform(X)

**Feature decomposition using Principal Component Analysis (PCA)**

From the pair plot in **Part\_2**, lot of feature pairs divide nicely the data to a similar extent, therefore, it makes sense to use one of the dimensionality reduction methods to try to use as many features as possible and maintian as much information as possible when working with only 2 dimensions. I will use PCA.

In [34]:

**from** sklearn.decomposition **import** PCA

*# feature extraction*

pca **=** PCA(n\_components**=**10)

fit **=** pca**.**fit(Xs)

*# summarize components*

*#print("Explained Variance: %s") % fit.explained\_variance\_ratio\_*

*#print(fit.components\_)*

In [35]:

X\_pca **=** pca**.**transform(Xs)

PCA\_df **=** pd**.**DataFrame()

PCA\_df['PCA\_1'] **=** X\_pca[:,0]

PCA\_df['PCA\_2'] **=** X\_pca[:,1]

plt**.**plot(PCA\_df['PCA\_1'][df**.**diagnosis **==** 'M'],PCA\_df['PCA\_2'][df**.**diagnosis **==** 'M'],'o', alpha **=** 0.7, color **=** 'r')

plt**.**plot(PCA\_df['PCA\_1'][df**.**diagnosis **==** 'B'],PCA\_df['PCA\_2'][df**.**diagnosis **==** 'B'],'o', alpha **=** 0.7, color **=** 'b')

plt**.**xlabel('PCA\_1')

plt**.**ylabel('PCA\_2')

plt**.**legend(['Malignant','Benign'])

plt**.**show()

Now, what we got after applying the linear PCA transformation is a lower dimensional subspace (from 3D to 2D in this case), where the samples are **most spread** along the new feature axes.

In [36]:

*#The amount of variance that each PC explains*

var**=** pca**.**explained\_variance\_ratio\_

*#Cumulative Variance explains*

*#var1=np.cumsum(np.round(pca.explained\_variance\_ratio\_, decimals=4)\*100)*

*#print(var1)*

**Deciding How Many Principal Components to Retain**

In order to decide how many principal components should be retained, it is common to summarise the results of a principal components analysis by making a scree plot. More about scree plot can be found [**here**](http://python-for-multivariate-analysis.readthedocs.io/a_little_book_of_python_for_multivariate_analysis.html), and [**hear**](https://www.analyticsvidhya.com/blog/2016/03/practical-guide-principal-component-analysis-python/).

In [37]:

*#The amount of variance that each PC explains*

var**=** pca**.**explained\_variance\_ratio\_

*#Cumulative Variance explains*

*#var1=np.cumsum(np.round(pca.explained\_variance\_ratio\_, decimals=4)\*100)*

*#print(var1)*

plt**.**plot(var)

plt**.**title('Scree Plot')

plt**.**xlabel('Principal Component')

plt**.**ylabel('Eigenvalue')

leg **=** plt**.**legend(['Eigenvalues from PCA'], loc**=**'best', borderpad**=**0.3, shadow**=False**, markerscale**=**0.4)

leg**.**get\_frame()**.**set\_alpha(0.4)

leg**.**set\_draggable(state**=True**)

plt**.**show()

**Observation**

The most obvious change in slope in the scree plot occurs at component 2, which is the “elbow” of the scree plot. Therefore, it cound be argued based on the basis of the scree plot that the first three components should be retained.

**Part\_3 Summary: Data Preprocing Approach used**

1. assign features to a NumPy array X, and transform the class labels from their original string representation (M and B) into integers
2. Split data into training and test sets
3. Standardize the data.
4. Obtain the Eigenvectors and Eigenvalues from the covariance matrix or correlation matrix
5. Sort eigenvalues in descending order and choose the kk eigenvectors that correspond to the kk largest eigenvalues where k is the number of dimensions of the new feature subspace (�≤��≤�).
6. Construct the projection matrix W from the selected k eigenvectors.
7. Transform the original dataset X via W to obtain a k-dimensional feature subspace Y.

It is common to select a subset of features that have the largest correlation with the class labels. The effect of feature selection must be assessed within a complete modeling pipeline in order to give you an unbiased estimated of your model's true performance. Hence, in the next section you will first be introduced to cross-validation, before applying the PCA-based feature selection strategy in the model building pipeline.

**Part\_4: Predictive model using Support Vector Machine (SVM)**

Support vector machines (SVMs) learning algorithm will be used to build the predictive model. SVMs are one of the most popular classification algorithms, and have an elegant way of transforming nonlinear data so that one can use a linear algorithm to fit a linear model to the data (Cortes and Vapnik 1995).

Kernelized support vector machines are powerful models and perform well on a variety of datasets.

1. SVMs allow for complex decision boundaries, even if the data has only a few features.
2. They work well on low-dimensional and high-dimensional data (i.e., few and many features), but don’t scale very well with the number of samples.

**Running an SVM on data with up to 10,000 samples might work well, but working with datasets of size 100,000 or more can become challenging in terms of runtime and memory usage.**

1. SVMs requires careful preprocessing of the data and tuning of the parameters. This is why, these days, most people instead use tree-based models such as random forests or gradient boosting (which require little or no preprocessing) in many applications.
2. SVM models are hard to inspect; it can be difficult to understand why a particular prediction was made, and it might be tricky to explain the model to a nonexpert.

**Important Parameters**

The important parameters in kernel SVMs are the

* Regularization parameter C,
* The choice of the kernel,(linear, radial basis function(RBF) or polynomial)
* Kernel-specific parameters.

gamma and C both control the complexity of the model, with large values in either resulting in a more complex model. Therefore, good settings for the two parameters are usually strongly correlated, and C and gamma should be adjusted together.

In [38]:

**%matplotlib** inline

**import** matplotlib.pyplot **as** plt

*#Load libraries for data processing*

**import** pandas **as** pd *#data processing, CSV file I/O (e.g. pd.read\_csv)*

**import** numpy **as** np

**from** scipy.stats **import** norm

*## Supervised learning.*

**from** sklearn.preprocessing **import** StandardScaler

**from** sklearn.preprocessing **import** LabelEncoder

**from** sklearn.model\_selection **import** train\_test\_split

**from** sklearn.svm **import** SVC

**from** sklearn.model\_selection **import** cross\_val\_score

**from** sklearn.pipeline **import** make\_pipeline

**from** sklearn.metrics **import** confusion\_matrix

**from** sklearn **import** metrics, preprocessing

**from** sklearn.metrics **import** classification\_report

*# visualization*

**import** seaborn **as** sns

plt**.**style**.**use('fivethirtyeight')

sns**.**set\_style("white")

plt**.**rcParams['figure.figsize'] **=** (8,4)

*#plt.rcParams['axes.titlesize'] = 'large'*

In [39]:

df **=** pd**.**read\_csv('data/data\_clean.csv', index\_col**=False**)

df**.**drop('Unnamed: 0',axis**=**1, inplace**=True**)

df**.**head(3)

Out[39]:

|  | **diagnosis** | **radius\_mean** | **texture\_mean** | **perimeter\_mean** | **area\_mean** | **smoothness\_mean** | **compactness\_mean** | **concavity\_mean** | **concave points\_mean** | **symmetry\_mean** | **...** | **radius\_worst** | **texture\_worst** | **perimeter\_worst** | **area\_worst** | **smoothness\_worst** | **compactness\_worst** | **concavity\_worst** | **concave points\_worst** | **symmetry\_worst** | **fractal\_dimension\_worst** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | M | 17.99 | 10.38 | 122.8 | 1001.0 | 0.11840 | 0.27760 | 0.3001 | 0.14710 | 0.2419 | ... | 25.38 | 17.33 | 184.6 | 2019.0 | 0.1622 | 0.6656 | 0.7119 | 0.2654 | 0.4601 | 0.11890 |
| **1** | M | 20.57 | 17.77 | 132.9 | 1326.0 | 0.08474 | 0.07864 | 0.0869 | 0.07017 | 0.1812 | ... | 24.99 | 23.41 | 158.8 | 1956.0 | 0.1238 | 0.1866 | 0.2416 | 0.1860 | 0.2750 | 0.08902 |
| **2** | M | 19.69 | 21.25 | 130.0 | 1203.0 | 0.10960 | 0.15990 | 0.1974 | 0.12790 | 0.2069 | ... | 23.57 | 25.53 | 152.5 | 1709.0 | 0.1444 | 0.4245 | 0.4504 | 0.2430 | 0.3613 | 0.08758 |

3 rows × 31 columns

In [40]:

*#Assign predictors to a variable of ndarray (matrix) type*

array **=** df**.**values

X **=** array[:,1:31] *# features*

y **=** array[:,0]

*#transform the class labels from their original string representation (M and B) into integers*

le **=** LabelEncoder()

y **=** le**.**fit\_transform(y)

*# Normalize the data (center around 0 and scale to remove the variance).*

scaler **=** StandardScaler()

Xs **=** scaler**.**fit\_transform(X)

**Classification with cross-validation**

As discussed in **Part\_3** splitting the data into test and training sets is crucial to avoid overfitting. This allows generalization of real, previously-unseen data. Cross-validation extends this idea further. Instead of having a single train/test split, we specify **so-called folds** so that the data is divided into similarly-sized folds.

* Training occurs by taking all folds except one – referred to as the holdout sample.
* On the completion of the training, you test the performance of your fitted model using the holdout sample.
* The holdout sample is then thrown back with the rest of the other folds, and a different fold is pulled out as the new holdout sample.
* Training is repeated again with the remaining folds and we measure performance using the holdout sample. This process is repeated until each fold has had a chance to be a test or holdout sample.
* The expected performance of the classifier, called cross-validation error, is then simply an average of error rates computed on each holdout sample.

This process is demonstrated by first performing a standard train/test split, and then computing cross-validation error.

In [41]:

*# Divide records in training and testing sets.*

X\_train, X\_test, y\_train, y\_test **=** train\_test\_split(Xs, y, test\_size**=**0.3, random\_state**=**2, stratify**=**y)

*# Create an SVM classifier and train it on 70% of the data set.*

clf **=** SVC(probability**=True**)

clf**.**fit(X\_train, y\_train)

*# Analyze accuracy of predictions on 30% of the holdout test sample.*

classifier\_score **=** clf**.**score(X\_test, y\_test)

print ('\n➔ The classifier accuracy score is {:03.2f}\n'**.**format(classifier\_score))

➔ The classifier accuracy score is 0.95

To get a better measure of prediction accuracy (which you can use as a proxy for “goodness of fit” of the model), you can successively split the data into folds that you will use for training and testing:

In [42]:

*# Get average of 3-fold cross-validation score using an SVC estimator.*

n\_folds **=** 3

cv\_error **=** np**.**average(cross\_val\_score(SVC(), Xs, y, cv**=**n\_folds))

print ('\n➔ The {}-fold cross-validation accuracy score for this classifier is {:.2f}\n'**.**format(n\_folds, cv\_error))

➔ The 3-fold cross-validation accuracy score for this classifier is 0.97

The above evaluations were based on using the entire set of features. You will now employ the correlation-based feature selection strategy to assess the effect of using 3 features which have the best correlation with the class labels.

In [43]:

**from** sklearn.feature\_selection **import** SelectKBest, f\_regression

clf2 **=** make\_pipeline(SelectKBest(f\_regression, k**=**3),SVC(probability**=True**))

scores **=** cross\_val\_score(clf2, Xs, y, cv**=**3)

*# Get average of 3-fold cross-validation score using an SVC estimator.*

n\_folds **=** 3

cv\_error **=** np**.**average(cross\_val\_score(SVC(), Xs, y, cv**=**n\_folds))

print ('\n➔ The {}-fold cross-validation accuracy score for this classifier is {:.2f}\n'**.**format(n\_folds, cv\_error))

➔ The 3-fold cross-validation accuracy score for this classifier is 0.97

In [44]:

print (scores)

avg **=** (100**\***np**.**mean(scores), 100**\***np**.**std(scores)**/**np**.**sqrt(scores**.**shape[0]))

print ("➔ Average score and uncertainty: (%.2f +- %.3f)%%"**%avg**)

[0.93157895 0.95263158 0.94179894]

➔ Average score and uncertainty: (94.20 +- 0.496)%

From the above results, you can see that only a fraction of the features are required to build a model that performs similarly to models based on using the entire set of features. Feature selection is an important part of the model-building process that you must always pay particular attention to. The details are beyond the scope of this notebook. In the rest of the analysis, you will continue using the entire set of features.

**Model Accuracy: Receiver Operating Characteristic (ROC) curve**

In statistical modeling and machine learning, a commonly-reported performance measure of model accuracy for binary classification problems is Area Under the Curve (AUC).

To understand what information the ROC curve conveys, consider the so-called confusion matrix that essentially is a two-dimensional table where the classifier model is on one axis (vertical), and ground truth is on the other (horizontal) axis, as shown below. Either of these axes can take two values (as depicted)

| **Model says "+"** | **Model says "-"** |
| --- | --- |
| True positive | False negative | \*\* Actual: "+" \*\* |
| False positive | True negative | Actual: "-" |

In an ROC curve, you plot “True Positive Rate” on the Y-axis and “False Positive Rate” on the X-axis, where the values “true positive”, “false negative”, “false positive”, and “true negative” are events (or their probabilities) as described above. The rates are defined according to the following:

* True positive rate (or sensitivity)}: tpr = tp / (tp + fn)
* False positive rate: fpr = fp / (fp + tn)
* True negative rate (or specificity): tnr = tn / (fp + tn)

In all definitions, the denominator is a row margin in the above confusion matrix. Thus,one can express

* the true positive rate (tpr) as the probability that the model says "+" when the real value is indeed "+" (i.e., a conditional probability). However, this does not tell you how likely you are to be correct when calling "+" (i.e., the probability of a true positive, conditioned on the test result being "+").

In [45]:

*# The confusion matrix helps visualize the performance of the algorithm.*

y\_pred **=** clf**.**fit(X\_train, y\_train)**.**predict(X\_test)

cm **=** metrics**.**confusion\_matrix(y\_test, y\_pred)

*#print(cm)*

In [46]:

**%matplotlib** inline

**import** matplotlib.pyplot **as** plt

**from** IPython.display **import** Image, display

fig, ax **=** plt**.**subplots(figsize**=**(5, 5))

ax**.**matshow(cm, cmap**=**plt**.**cm**.**Reds, alpha**=**0.3)

**for** i **in** range(cm**.**shape[0]):

**for** j **in** range(cm**.**shape[1]):

ax**.**text(x**=**j, y**=**i,

s**=**cm[i, j],

va**=**'center', ha**=**'center')

plt**.**xlabel('Predicted Values', )

plt**.**ylabel('Actual Values')

plt**.**show()

print(classification\_report(y\_test, y\_pred ))

precision recall f1-score support

0 0.94 0.99 0.96 107

1 0.98 0.89 0.93 64

accuracy 0.95 171

macro avg 0.96 0.94 0.95 171

weighted avg 0.95 0.95 0.95 171

**Observation**

There are two possible predicted classes: "1" and "0". Malignant = 1 (indicates prescence of cancer cells) and Benign = 0 (indicates abscence).

* The classifier made a total of 174 predictions (i.e 174 patients were being tested for the presence breast cancer).
* Out of those 174 cases, the classifier predicted "yes" 58 times, and "no" 113 times.
* In reality, 64 patients in the sample have the disease, and 107 patients do not.

**Rates as computed from the confusion matrix**

1. **Accuracy**: Overall, how often is the classifier correct?
   * (TP+TN)/total = (57+106)/171 = 0.95
2. **Misclassification Rate**: Overall, how often is it wrong?
   * (FP+FN)/total = (1+7)/171 = 0.05 equivalent to 1 minus Accuracy also known as \***"Error Rate"**\*
3. **True Positive Rate:** When it's actually yes, how often does it predict 1?
   * TP/actual yes = 57/64 = 0.89 also known as "Sensitivity" or \***"Recall"**\*
4. **False Positive Rate**: When it's actually 0, how often does it predict 1?
   * FP/actual no = 1/107 = 0.01
5. **Specificity**: When it's actually 0, how often does it predict 0? also know as **true positive rate**
   * TN/actual no = 106/107 = 0.99 equivalent to 1 minus False Positive Rate
6. **Precision**: When it predicts 1, how often is it correct?
   * TP/predicted yes = 57/58 = 0.98
7. **Prevalence**: How often does the yes condition actually occur in our sample?
   * actual yes/total = 64/171 = 0.34

In [47]:

**from** sklearn.metrics **import** roc\_curve, auc

*# Plot the receiver operating characteristic curve (ROC).*

plt**.**figure(figsize**=**(10,8))

probas\_ **=** clf**.**predict\_proba(X\_test)

fpr, tpr, thresholds **=** roc\_curve(y\_test, probas\_[:, 1])

roc\_auc **=** auc(fpr, tpr)

plt**.**plot(fpr, tpr, lw**=**1, label**=**'ROC fold (area = %0.2f)' **%** (roc\_auc))

plt**.**plot([0, 1], [0, 1], '--', color**=**(0.6, 0.6, 0.6), label**=**'Random')

plt**.**xlim([**-**0.05, 1.05])

plt**.**ylim([**-**0.05, 1.05])

plt**.**xlabel('False Positive Rate')

plt**.**ylabel('True Positive Rate')

plt**.**title('Receiver operating characteristic example')

plt**.**axes()**.**set\_aspect(1)

<ipython-input-47-2941f472df33>:15: MatplotlibDeprecationWarning: Adding an axes using the same arguments as a previous axes currently reuses the earlier instance. In a future version, a new instance will always be created and returned. Meanwhile, this warning can be suppressed, and the future behavior ensured, by passing a unique label to each axes instance.

plt.axes().set\_aspect(1)

* To interpret the ROC correctly, consider what the points that lie along the diagonal represent. For these situations, there is an equal chance of "+" and "-" happening. Therefore, this is not that different from making a prediction by tossing of an unbiased coin. Put simply, the classification model is random.
* For the points above the diagonal, tpr > fpr, and the model says that you are in a zone where you are performing better than random. For example, assume tpr = 0.99 and fpr = 0.01, Then, the probability of being in the true positive group is (0.99/(0.99+0.01))=99%. Furthermore, holding fpr constant, it is easy to see that the more vertically above the diagonal you are positioned, the better the classification model.

**Next we'll look into optimizing the class**

[**Cortes, Corinna, and Vladimir Vapnik. 1995. 'Support-Vector Networks.' Machine Learning 20: 273. Accessed September 3, 2016. doi: 10.1023/A:1022627411411**](https://link.springer.com/article/10.1007/BF00994018).

**Part\_5: Optimizing the SVM Classifier**

Machine learning models are parameterized so that their behavior can be tuned for a given problem. Models can have many parameters and finding the best combination of parameters can be treated as a search problem. In this part, we'll aim to tune parameters of the SVM Classification model using scikit-learn.

**Load Libraries and Data**

In [48]:

**%matplotlib** inline

**import** matplotlib.pyplot **as** plt

*#Load libraries for data processing*

**import** pandas **as** pd *#data processing, CSV file I/O (e.g. pd.read\_csv)*

**import** numpy **as** np

**from** scipy.stats **import** norm

*## Supervised learning.*

**from** sklearn.preprocessing **import** StandardScaler

**from** sklearn.preprocessing **import** LabelEncoder

**from** sklearn.model\_selection **import** train\_test\_split

**from** sklearn.svm **import** SVC

**from** sklearn.model\_selection **import** cross\_val\_score

**from** sklearn.model\_selection **import** GridSearchCV

**from** sklearn.pipeline **import** make\_pipeline

**from** sklearn.metrics **import** confusion\_matrix

**from** sklearn **import** metrics, preprocessing

**from** sklearn.metrics **import** classification\_report

**from** sklearn.feature\_selection **import** SelectKBest, f\_regression

*# visualization*

**import** seaborn **as** sns

plt**.**style**.**use('fivethirtyeight')

sns**.**set\_style("white")

plt**.**rcParams['figure.figsize'] **=** (8,4)

*#plt.rcParams['axes.titlesize'] = 'large'*

**Build a predictive model and evaluate with 5-cross validation using support vector classifies (ref Part\_4) for details**

In [49]:

X

Out[49]:

array([[17.99, 10.38, 122.8, ..., 0.2654, 0.4601, 0.1189],

[20.57, 17.77, 132.9, ..., 0.186, 0.275, 0.08902],

[19.69, 21.25, 130.0, ..., 0.243, 0.3613, 0.08758],

...,

[16.6, 28.08, 108.3, ..., 0.1418, 0.2218, 0.0782],

[20.6, 29.33, 140.1, ..., 0.265, 0.4087, 0.124],

[7.76, 24.54, 47.92, ..., 0.0, 0.2871, 0.07039]], dtype=object)

In [50]:

df **=** pd**.**read\_csv('data/data\_clean.csv', index\_col**=False**)

df**.**drop('Unnamed: 0',axis**=**1, inplace**=True**)

*# Assign predictors to a variable of ndarray (matrix) type*

array **=** df**.**values

X **=** array[:,1:31]

y **=** array[:,0]

*# Transform the class labels from their original string representation (M and B) into integers*

le **=** LabelEncoder()

y **=** le**.**fit\_transform(y)

*# Normalize the data (center around 0 and scale to remove the variance).*

scaler **=**StandardScaler()

Xs **=** scaler**.**fit\_transform(X)

**from** sklearn.decomposition **import** PCA

*# feature extraction*

pca **=** PCA(n\_components**=**10)

fit **=** pca**.**fit(Xs)

X\_pca **=** pca**.**transform(Xs)

*# Divide records in training and testing sets.*

X\_train, X\_test, y\_train, y\_test **=** train\_test\_split(X\_pca, y, test\_size**=**0.3, random\_state**=**2, stratify**=**y)

*# Create an SVM classifier and train it on 70% of the data set.*

clf **=** SVC(probability**=True**)

clf**.**fit(X\_train, y\_train)

*# Analyze accuracy of predictions on 30% of the holdout test sample.*

classifier\_score **=** clf**.**score(X\_test, y\_test)

print ('\nThe classifier accuracy score is {:03.2f}\n'**.**format(classifier\_score))

clf2 **=** make\_pipeline(SelectKBest(f\_regression, k**=**3),SVC(probability**=True**))

scores **=** cross\_val\_score(clf2, X\_pca, y, cv**=**3)

*# Get average of 5-fold cross-validation score using an SVC estimator.*

n\_folds **=** 5

cv\_error **=** np**.**average(cross\_val\_score(SVC(), X\_pca, y, cv**=**n\_folds))

*#print ('\nThe {}-fold cross-validation accuracy score for this classifier is {:.2f}\n'.format(n\_folds, cv\_error))*

y\_pred **=** clf**.**fit(X\_train, y\_train)**.**predict(X\_test)

cm **=** metrics**.**confusion\_matrix(y\_test, y\_pred)

print(classification\_report(y\_test, y\_pred ))

fig, ax **=** plt**.**subplots(figsize**=**(5, 5))

ax**.**matshow(cm, cmap**=**plt**.**cm**.**Reds, alpha**=**0.3)

**for** i **in** range(cm**.**shape[0]):

**for** j **in** range(cm**.**shape[1]):

ax**.**text(x**=**j, y**=**i,

s**=**cm[i, j],

va**=**'center', ha**=**'center')

plt**.**xlabel('Predicted Values', )

plt**.**ylabel('Actual Values')

plt**.**show()

The classifier accuracy score is 0.96

precision recall f1-score support

0 0.95 0.99 0.97 107

1 0.98 0.91 0.94 64

accuracy 0.96 171

macro avg 0.96 0.95 0.96 171

weighted avg 0.96 0.96 0.96 171

**Importance of optimizing a classifier**

We can tune two key parameters of the SVM algorithm:

* the value of C (how much to relax the margin)
* and the type of kernel.

The default for SVM (the SVC class) is to use the Radial Basis Function (RBF) kernel with a C value set to 1.0. Like with KNN, we will perform a grid search using 10-fold cross validation with a standardized copy of the training dataset. We will try a number of simpler kernel types and C values with less bias and more bias (less than and more than 1.0 respectively).

Python scikit-learn provides two simple methods for algorithm parameter tuning:

* Grid Search Parameter Tuning.
* Random Search Parameter Tuning.

In [51]:

*# Train classifiers.*

kernel\_values **=** [ 'linear' , 'poly' , 'rbf' , 'sigmoid' ]

param\_grid **=** {'C': np**.**logspace(**-**3, 2, 6), 'gamma': np**.**logspace(**-**3, 2, 6),'kernel': kernel\_values}

grid **=** GridSearchCV(SVC(), param\_grid**=**param\_grid, cv**=**5)

grid**.**fit(X\_train, y\_train)

Out[51]:

GridSearchCV(cv=5, estimator=SVC(),

param\_grid={'C': array([1.e-03, 1.e-02, 1.e-01, 1.e+00, 1.e+01, 1.e+02]),

'gamma': array([1.e-03, 1.e-02, 1.e-01, 1.e+00, 1.e+01, 1.e+02]),

'kernel': ['linear', 'poly', 'rbf', 'sigmoid']})

In [52]:

print("The best parameters are %s with a score of %0.2f"

**%** (grid**.**best\_params\_, grid**.**best\_score\_))

The best parameters are {'C': 0.1, 'gamma': 0.001, 'kernel': 'linear'} with a score of 0.98

In [53]:

grid**.**best\_estimator\_**.**probability **=** **True**

clf **=** grid**.**best\_estimator\_

In [54]:

y\_pred **=** clf**.**fit(X\_train, y\_train)**.**predict(X\_test)

cm **=** metrics**.**confusion\_matrix(y\_test, y\_pred)

*#print(cm)*

print(classification\_report(y\_test, y\_pred ))

fig, ax **=** plt**.**subplots(figsize**=**(5, 5))

ax**.**matshow(cm, cmap**=**plt**.**cm**.**Reds, alpha**=**0.3)

**for** i **in** range(cm**.**shape[0]):

**for** j **in** range(cm**.**shape[1]):

ax**.**text(x**=**j, y**=**i,

s**=**cm[i, j],

va**=**'center', ha**=**'center')

plt**.**xlabel('Predicted Values', )

plt**.**ylabel('Actual Values')

plt**.**show()

precision recall f1-score support

0 0.96 1.00 0.98 107

1 1.00 0.92 0.96 64

accuracy 0.97 171

macro avg 0.98 0.96 0.97 171

weighted avg 0.97 0.97 0.97 171

**Decision boundaries of different classifiers**

Let's see the decision boundaries produced by the linear, Gaussian and polynomial classifiers.

In [55]:

**import** matplotlib.pyplot **as** plt

**from** matplotlib.colors **import** ListedColormap

**from** sklearn **import** svm, datasets

**def** decision\_plot(X\_train, y\_train, n\_neighbors, weights):

h **=** .02 *# step size in the mesh*

Xtrain **=** X\_train[:, :2] *# we only take the first two features.*

*#================================================================*

*# Create color maps*

*#================================================================*

cmap\_light **=** ListedColormap(['#FFAAAA', '#AAFFAA', '#AAAAFF'])

cmap\_bold **=** ListedColormap(['#FF0000', '#00FF00', '#0000FF'])

*#================================================================*

*# we create an instance of SVM and fit out data.*

*# We do not scale ourdata since we want to plot the support vectors*

*#================================================================*

C **=** 1.0 *# SVM regularization parameter*

svm **=** SVC(kernel**=**'linear', random\_state**=**0, gamma**=**0.1, C**=**C)**.**fit(Xtrain, y\_train)

rbf\_svc **=** SVC(kernel**=**'rbf', gamma**=**0.7, C**=**C)**.**fit(Xtrain, y\_train)

poly\_svc **=** SVC(kernel**=**'poly', degree**=**3, C**=**C)**.**fit(Xtrain, y\_train)

In [56]:

**%matplotlib** inline

plt**.**rcParams['figure.figsize'] **=** (15, 9)

plt**.**rcParams['axes.titlesize'] **=** 'large'

*# create a mesh to plot in*

x\_min, x\_max **=** Xtrain[:, 0]**.**min() **-** 1, Xtrain[:, 0]**.**max() **+** 1

y\_min, y\_max **=** Xtrain[:, 1]**.**min() **-** 1, Xtrain[:, 1]**.**max() **+** 1

xx, yy **=** np**.**meshgrid(np**.**arange(x\_min, x\_max, 0.1),

np**.**arange(y\_min, y\_max, 0.1))

*# title for the plots*

titles **=** ['SVC with linear kernel',

'SVC with RBF kernel',

'SVC with polynomial (degree 3) kernel']

In [57]:

**for** i, clf **in** enumerate((svm, rbf\_svc, poly\_svc)):

*# Plot the decision boundary. For that, we will assign a color to each*

*# point in the mesh [x\_min, x\_max]x[y\_min, y\_max].*

plt**.**subplot(2, 2, i **+** 1)

plt**.**subplots\_adjust(wspace**=**0.4, hspace**=**0.4)

Z **=** clf**.**predict(np**.**c\_[xx**.**ravel(), yy**.**ravel()])

*# Put the result into a color plot*

Z **=** Z**.**reshape(xx**.**shape)

plt**.**contourf(xx, yy, Z, cmap**=**plt**.**cm**.**coolwarm, alpha**=**0.8)

*# Plot also the training points*

plt**.**scatter(Xtrain[:, 0], Xtrain[:, 1], c**=**y\_train, cmap**=**plt**.**cm**.**coolwarm)

plt**.**xlabel('radius\_mean')

plt**.**ylabel('texture\_mean')

plt**.**xlim(xx**.**min(), xx**.**max())

plt**.**ylim(yy**.**min(), yy**.**max())

plt**.**xticks(())

plt**.**yticks(())

plt**.**title(titles[i])

plt**.**show()

**Conclusion**

This work demonstrates the modelling of breast cancer as classification task using Support Vector Machine

The SVM performs better when the dataset is standardized so that all attributes have a mean value of zero and a standard deviation of one. We can calculate this from the entire training dataset and apply the same transform to the input attributes from the validation dataset.

**Next Task:**

1. Summary and conclusion of findings
2. Compare with other classification methods
   * Decision trees with **tree.DecisionTreeClassifier()**
   * K-nearest neighbors with **neighbors.KNeighborsClassifier()**
   * Random forests with **ensemble.RandomForestClassifier()**
   * Perceptron (both gradient and stochastic gradient) with **mlxtend.classifier.Perceptron**
   * Multilayer perceptron network (both gradient and stochastic gradient) with **mlxtend.classifier.MultiLayerPerceptron**

**Part\_6: Automate the ML process using pipelines**

There are standard workflows in a machine learning project that can be automated. In Python scikit-learn, Pipelines help to clearly define and automate these workflows.

* Pipelines help overcome common problems like data leakage in your test harness.
* Python scikit-learn provides a Pipeline utility to help automate machine learning workflows.
* Pipelines work by allowing for a linear sequence of data transforms to be chained together culminating in a modeling process that can be evaluated.

**Data Preparation and Modeling Pipeline**

In [1]:

**%matplotlib** inline

**import** matplotlib.pyplot **as** plt

*# Create a pipeline that standardizes the data then creates a model*

*#Load libraries for data processing*

**import** pandas **as** pd *#data processing, CSV file I/O (e.g. pd.read\_csv)*

**import** numpy **as** np

**from** scipy.stats **import** norm

**from** sklearn.model\_selection **import** train\_test\_split

**from** sklearn.model\_selection **import** cross\_val\_score

**from** sklearn.model\_selection **import** KFold

**from** sklearn.preprocessing **import** LabelEncoder

**from** sklearn.preprocessing **import** StandardScaler

**from** sklearn.decomposition **import** PCA

**from** sklearn.pipeline **import** Pipeline

**from** sklearn.model\_selection **import** GridSearchCV

**from** sklearn.linear\_model **import** LogisticRegression

**from** sklearn.tree **import** DecisionTreeClassifier

**from** sklearn.neighbors **import** KNeighborsClassifier

**from** sklearn.discriminant\_analysis **import** LinearDiscriminantAnalysis

**from** sklearn.naive\_bayes **import** GaussianNB

**from** sklearn.svm **import** SVC

**from** sklearn.metrics **import** confusion\_matrix

**from** sklearn.metrics **import** accuracy\_score

**from** sklearn.metrics **import** classification\_report

*# visualization*

**import** seaborn **as** sns

plt**.**style**.**use('fivethirtyeight')

sns**.**set\_style("white")

plt**.**rcParams['figure.figsize'] **=** (8,4)

*#plt.rcParams['axes.titlesize'] = 'large'*

**Evaluate Some Algorithms**

Now it is time to create some models of the data and estimate their accuracy on unseen data. Here is what we are going to cover in this step:

1. Separate out a validation dataset.
2. Setup the test harness to use 10-fold cross validation.
3. Build 5 different models
4. Select the best model

**Dataset Validation**

In [2]:

*#load data*

df **=** pd**.**read\_csv('data/data\_clean.csv', index\_col**=False**)

df**.**drop('Unnamed: 0',axis**=**1, inplace**=True**)

*# Split-out validation dataset*

array **=** df**.**values

X **=** array[:,1:31]

y **=** array[:,0]

*# Divide records in training and testing sets.*

X\_train, X\_test, y\_train, y\_test **=** train\_test\_split(X, y, test\_size**=**0.3, random\_state**=**7)

*#transform the class labels from their original string representation (M and B) into integers*

le **=** LabelEncoder()

y **=** le**.**fit\_transform(y)

**Evaluate Algorithms: Baseline**

In [3]:

*# Spot-Check Algorithms*

models **=** []

models**.**append(('LR', LogisticRegression()))

models**.**append(('LDA', LinearDiscriminantAnalysis()))

models**.**append(('KNN', KNeighborsClassifier()))

models**.**append(('CART', DecisionTreeClassifier()))

models**.**append(('NB', GaussianNB()))

models**.**append(('SVM', SVC()))

*# Test options and evaluation metric*

num\_folds **=** 10

num\_instances **=** len(X\_train)

seed **=** 7

scoring **=** 'accuracy'

*# Test options and evaluation metric*

num\_folds **=** 10

num\_instances **=** len(X\_train)

seed **=** 7

scoring **=** 'accuracy'

results **=** []

names **=** []

**for** name, model **in** models:

kfold **=** KFold(n\_splits**=**num\_folds, random\_state**=**seed, shuffle**=True**)

cv\_results **=** cross\_val\_score(model, X\_train, y\_train, cv**=**kfold, scoring**=**scoring)

results**.**append(cv\_results)

names**.**append(name)

msg **=** "%s: %f (%f)" **%** (name, cv\_results**.**mean(), cv\_results**.**std())

print(msg)

print('\n➔ 10-Fold cross-validation accurcay score for the training data for six classifiers')

C:\ProgramData\Anaconda3\lib\site-packages\sklearn\linear\_model\\_logistic.py:763: ConvergenceWarning: lbfgs failed to converge (status=1):

STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max\_iter) or scale the data as shown in:

https://scikit-learn.org/stable/modules/preprocessing.html

Please also refer to the documentation for alternative solver options:

https://scikit-learn.org/stable/modules/linear\_model.html#logistic-regression

n\_iter\_i = \_check\_optimize\_result(

C:\ProgramData\Anaconda3\lib\site-packages\sklearn\linear\_model\\_logistic.py:763: ConvergenceWarning: lbfgs failed to converge (status=1):

STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max\_iter) or scale the data as shown in:

https://scikit-learn.org/stable/modules/preprocessing.html

Please also refer to the documentation for alternative solver options:

https://scikit-learn.org/stable/modules/linear\_model.html#logistic-regression

n\_iter\_i = \_check\_optimize\_result(

C:\ProgramData\Anaconda3\lib\site-packages\sklearn\linear\_model\\_logistic.py:763: ConvergenceWarning: lbfgs failed to converge (status=1):

STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max\_iter) or scale the data as shown in:

https://scikit-learn.org/stable/modules/preprocessing.html

Please also refer to the documentation for alternative solver options:

https://scikit-learn.org/stable/modules/linear\_model.html#logistic-regression

n\_iter\_i = \_check\_optimize\_result(

C:\ProgramData\Anaconda3\lib\site-packages\sklearn\linear\_model\\_logistic.py:763: ConvergenceWarning: lbfgs failed to converge (status=1):

STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max\_iter) or scale the data as shown in:

https://scikit-learn.org/stable/modules/preprocessing.html

Please also refer to the documentation for alternative solver options:

https://scikit-learn.org/stable/modules/linear\_model.html#logistic-regression

n\_iter\_i = \_check\_optimize\_result(

C:\ProgramData\Anaconda3\lib\site-packages\sklearn\linear\_model\\_logistic.py:763: ConvergenceWarning: lbfgs failed to converge (status=1):

STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max\_iter) or scale the data as shown in:

https://scikit-learn.org/stable/modules/preprocessing.html

Please also refer to the documentation for alternative solver options:

https://scikit-learn.org/stable/modules/linear\_model.html#logistic-regression

n\_iter\_i = \_check\_optimize\_result(

C:\ProgramData\Anaconda3\lib\site-packages\sklearn\linear\_model\\_logistic.py:763: ConvergenceWarning: lbfgs failed to converge (status=1):

STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max\_iter) or scale the data as shown in:

https://scikit-learn.org/stable/modules/preprocessing.html

Please also refer to the documentation for alternative solver options:

https://scikit-learn.org/stable/modules/linear\_model.html#logistic-regression

n\_iter\_i = \_check\_optimize\_result(

C:\ProgramData\Anaconda3\lib\site-packages\sklearn\linear\_model\\_logistic.py:763: ConvergenceWarning: lbfgs failed to converge (status=1):

STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max\_iter) or scale the data as shown in:

https://scikit-learn.org/stable/modules/preprocessing.html

Please also refer to the documentation for alternative solver options:

https://scikit-learn.org/stable/modules/linear\_model.html#logistic-regression

n\_iter\_i = \_check\_optimize\_result(

C:\ProgramData\Anaconda3\lib\site-packages\sklearn\linear\_model\\_logistic.py:763: ConvergenceWarning: lbfgs failed to converge (status=1):

STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max\_iter) or scale the data as shown in:

https://scikit-learn.org/stable/modules/preprocessing.html

Please also refer to the documentation for alternative solver options:

https://scikit-learn.org/stable/modules/linear\_model.html#logistic-regression

n\_iter\_i = \_check\_optimize\_result(

C:\ProgramData\Anaconda3\lib\site-packages\sklearn\linear\_model\\_logistic.py:763: ConvergenceWarning: lbfgs failed to converge (status=1):

STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max\_iter) or scale the data as shown in:

https://scikit-learn.org/stable/modules/preprocessing.html

Please also refer to the documentation for alternative solver options:

https://scikit-learn.org/stable/modules/linear\_model.html#logistic-regression

n\_iter\_i = \_check\_optimize\_result(

C:\ProgramData\Anaconda3\lib\site-packages\sklearn\linear\_model\\_logistic.py:763: ConvergenceWarning: lbfgs failed to converge (status=1):

STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max\_iter) or scale the data as shown in:

https://scikit-learn.org/stable/modules/preprocessing.html

Please also refer to the documentation for alternative solver options:

https://scikit-learn.org/stable/modules/linear\_model.html#logistic-regression

n\_iter\_i = \_check\_optimize\_result(

LR: 0.944808 (0.026834)

LDA: 0.954744 (0.018784)

KNN: 0.937179 (0.028218)

CART: 0.932244 (0.040281)

NB: 0.937115 (0.040822)

SVM: 0.901987 (0.047020)

➔ 10-Fold cross-validation accurcay score for the training data for six classifiers

In [4]:

len(X\_train)

Out[4]:

398

**Observation**

The results suggest That both Logistic Regression and LDA may be worth further study. These are just mean accuracy values. It is always wise to look at the distribution of accuracy values calculated across cross validation folds. We can do that graphically using box and whisker plots.

In [5]:

*# Compare Algorithms*

fig **=** plt**.**figure()

fig**.**suptitle('Algorithm Comparison')

ax **=** fig**.**add\_subplot(111)

plt**.**boxplot(results)

ax**.**set\_xticklabels(names)

plt**.**show()

**Observation**

The results show a similar tight distribution for all classifiers except SVM which is encouraging, suggesting low variance. The good results for SVM are satisfactory.

It is possible the varied distribution of the attributes may have an effect on the accuracy of algorithms such as SVM. In the next section we will repeat this spot-check with a standardized copy of the training dataset.

**2.1 Evaluate Algorithms: Standardize Data**

In [6]:

*# Standardize the dataset*

pipelines **=** []

pipelines**.**append(('ScaledLR', Pipeline([('Scaler', StandardScaler()),('LR', LogisticRegression())])))

pipelines**.**append(('ScaledLDA', Pipeline([('Scaler', StandardScaler()),('LDA', LinearDiscriminantAnalysis())])))

pipelines**.**append(('ScaledKNN', Pipeline([('Scaler', StandardScaler()),('KNN', KNeighborsClassifier())])))

pipelines**.**append(('ScaledCART', Pipeline([('Scaler', StandardScaler()),('CART', DecisionTreeClassifier())])))

pipelines**.**append(('ScaledNB', Pipeline([('Scaler', StandardScaler()),('NB', GaussianNB())])))

pipelines**.**append(('ScaledSVM', Pipeline([('Scaler', StandardScaler()),('SVM', SVC())])))

results **=** []

names **=** []

**for** name, model **in** pipelines:

kfold **=** KFold(n\_splits**=**num\_folds, random\_state**=**seed, shuffle**=True**)

cv\_results **=** cross\_val\_score(model, X\_train, y\_train, cv**=**kfold,

scoring**=**scoring)

results**.**append(cv\_results)

names**.**append(name)

msg **=** "%s: %f (%f)" **%** (name, cv\_results**.**mean(), cv\_results**.**std())

print(msg)

ScaledLR: 0.974936 (0.015813)

ScaledLDA: 0.954744 (0.018784)

ScaledKNN: 0.957372 (0.033665)

ScaledCART: 0.927244 (0.032326)

ScaledNB: 0.937115 (0.039261)

ScaledSVM: 0.967436 (0.027483)

In [7]:

*# Compare Algorithms*

fig **=** plt**.**figure()

fig**.**suptitle('Scaled Algorithm Comparison')

ax **=** fig**.**add\_subplot(111)

plt**.**boxplot(results)

ax**.**set\_xticklabels(names)

plt**.**show()

**Observations**

The results show that standardization of the data has lifted the skill of SVM to be the most accurate algorithm tested so far.

The results suggest digging deeper into the SVM and LDA and LR algorithms. It is very likely that configuration beyond the default may yield even more accurate models.

**3.0 Algorithm Tuning**

In this section we investigate tuning the parameters for three algorithms that show promise from the spot-checking in the previous section: LR, LDA and SVM.

**Tuning hyper-parameters - SVC estimator**

In [8]:

*#Make Support Vector Classifier Pipeline*

pipe\_svc **=** Pipeline([('scl', StandardScaler()),

('pca', PCA(n\_components**=**2)),

('clf', SVC(probability**=True**, verbose**=False**))])

*#Fit Pipeline to training Data*

pipe\_svc**.**fit(X\_train, y\_train)

*#print('➝ Fitted Pipeline to training Data')*

scores **=** cross\_val\_score(estimator**=**pipe\_svc, X**=**X\_train, y**=**y\_train, cv**=**10, n\_jobs**=**1, verbose**=**0)

print('➔ Model Training Accuracy: %.3f +/- %.3f' **%**(np**.**mean(scores), np**.**std(scores)))

*#Tune Hyperparameters*

param\_range **=** [0.0001, 0.001, 0.01, 0.1, 1.0, 10.0, 100.0, 1000.0]

param\_grid **=** [{'clf\_\_C': param\_range,'clf\_\_kernel': ['linear']},

{'clf\_\_C': param\_range,'clf\_\_gamma': param\_range,

'clf\_\_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('➔ Tuned Parameters Best Score: ',gs**.**best\_score\_)

print('➔ Best Parameters: \n',gs**.**best\_params\_)

➔ Model Training Accuracy: 0.940 +/- 0.034

➔ Tuned Parameters Best Score: 0.9446794871794871

➔ Best Parameters:

{'clf\_\_C': 1.0, 'clf\_\_kernel': 'linear'}

**Tuning the hyper-parameters: k-NN hyperparameters**

For your standard k-NN implementation, there are two primary hyperparameters that you’ll want to tune:

* The number of neighbors k.
* The distance metric/similarity function.

Both of these values can dramatically affect the accuracy of your k-NN classifier. Grid object is ready to do 10-fold cross validation on a KNN model using classification accuracy as the evaluation metric In addition, there is a parameter grid to repeat the 10-fold cross validation process 30 times Each time, the n\_neighbors parameter should be given a different value from the list We can't give **GridSearchCV** just a list We've to specify n\_neighbors should take on 1 through 30 You can set **n\_jobs = -1** to run computations in parallel (if supported by your computer and OS)

In [9]:

**from** sklearn.neighbors **import** KNeighborsClassifier **as** KNN

pipe\_knn **=** Pipeline([('scl', StandardScaler()),

('pca', PCA(n\_components**=**2)),

('clf', KNeighborsClassifier())])

*#Fit Pipeline to training Data*

pipe\_knn**.**fit(X\_train, y\_train)

scores **=** cross\_val\_score(estimator**=**pipe\_knn,

X**=**X\_train,

y**=**y\_train,

cv**=**10,

n\_jobs**=**1)

print('➝ Model Training Accuracy: %.3f +/- %.3f' **%**(np**.**mean(scores), np**.**std(scores)))

*#Tune Hyperparameters*

param\_range **=** range(1, 31)

param\_grid **=** [{'clf\_\_n\_neighbors': param\_range}]

*# instantiate the grid*

gs **=** GridSearchCV(estimator**=**pipe\_knn,

param\_grid**=**param\_grid,

cv**=**10,

scoring**=**'accuracy')

gs **=** gs**.**fit(X\_train, y\_train)

print('➔ Tuned Parameters Best Score: ',gs**.**best\_score\_)

print('➔ Best Parameters: \n',gs**.**best\_params\_)

➝ Model Training Accuracy: 0.927 +/- 0.044

➔ Tuned Parameters Best Score: 0.9396153846153847

➔ Best Parameters:

{'clf\_\_n\_neighbors': 19}

**Finalize Model**

In [10]:

*#Use best parameters*

clf\_svc **=** gs**.**best\_estimator\_

*#Get Final Scores*

clf\_svc**.**fit(X\_train, y\_train)

scores **=** cross\_val\_score(estimator**=**clf\_svc,

X**=**X\_train,

y**=**y\_train,

cv**=**10,

n\_jobs**=**1)

print('➔ Final Model Training Accuracy: %.3f +/- %.3f' **%**(np**.**mean(scores), np**.**std(scores)))

print('➜ Final Accuracy on Test set: %.5f' **%** clf\_svc**.**score(X\_test,y\_test))

➔ Final Model Training Accuracy: 0.940 +/- 0.034

➜ Final Accuracy on Test set: 0.94737

In [11]:

clf\_svc**.**fit(X\_train, y\_train)

y\_pred **=** clf\_svc**.**predict(X\_test)

print(accuracy\_score(y\_test, y\_pred))

print(confusion\_matrix(y\_test, y\_pred))

print(classification\_report(y\_test, y\_pred))

0.9473684210526315

[[113 3]

[ 6 49]]

precision recall f1-score support

B 0.95 0.97 0.96 116

M 0.94 0.89 0.92 55

accuracy 0.95 171

macro avg 0.95 0.93 0.94 171

weighted avg 0.95 0.95 0.95 171

**Summary**

Worked through a classification predictive modeling machine learning problem from end-to-end using Python. Specifically, the steps covered were:

1. Problem Definition (Breast Cancer data).
2. Loading the Dataset.
3. Analyze Data (same scale but di↵erent distributions of data).
   * Evaluate Algorithms (KNN looked good).
   * Evaluate Algorithms with Standardization (KNN and SVM looked good).
4. Algorithm Tuning (K=19 for KNN was good, SVM with an RBF kernel and C=100 was best)..
5. Finalize Model (use all training data and confirm using validation dataset)

In [ ]: