Cheat Sheet :

```
# importing the dataset
dataset = pd.read_csv('    .csv').[dataset.iloc[:, 2:4].values]
```

Missing Data values

```
from sklearn.preprocessing import Imputer
imputer = Imputer(missing_values = 'NaN', strategy = 'mean',
                  axis = 0)
                    └→ Impute column, 1. means → row
imputer = imputer.fit(x)
x = imputer.transform(x)
```

Categorical Data

```
from sklearn.preprocessing import LabelEncoder, OneHotEncoder
labelencoder_x = LabelEncoder()
X = labelencoder_x.fit_transform(x)[x[:, 0]]
onehotencoder = OneHotEncoder(categorical_features = [0])
X = onehotencoder.fit_transform(x).toarray()
```

Splitting the dataset into Trainingset and Testset :

```
from sklearn.cross_validation import train_test_split
X_train, X_test, Y_train, Y_test = train_test_split
                    (X, Y, test_size = 0.2, random_state = 0)
```

Feature Scaling : Two ways < Standardisation / Normalisation

```
from sklearn.preprocessing import StandardScaler
sc_X = StandardScaler()
X_train = sc_X.fit_transform(X_train)
X_test = sc_X.transform(X_test)
```

Linear Regression :

```
from sklearn.linear_model import LinearRegression
regressor = LinearRegression()
regressor.fit(x_train, y_train)
y_pred = regressor.predict(x_test)
```

Multiple linear Regression :

```
# encode the data if string exists (Label, OneHot-Encoding)
# Avoiding Dummy Variable Trap
    X = X[:, 1:]
# Fitting Multiple Linear Regression
from sklearn.linear_model import LinearRegression
regressor = LinearRegression()
regressor.fit(x_train, y_train)
y_pred = regressor.predict(x_test)
```

Matrix of Features I v's :

`[:, 0]` → create vector

But our Matrix of features should always be in matrices, not in vector. So `[:, 1:2]` → upper bound excluded.

Polynomial Regression :

```
from sklearn.preprocessing import PolynomialFeatures
poly_reg = PolynomialFeatures(degree = 2)   → creates a column of squared values.
X_poly = poly_reg.fit_transform(X)
lin_reg_2 = LinearRegression()
lin_reg_2.fit(X_poly, Y)
```

Visualizing the results :

```
plt.scatter(X, Y, color = 'red')
plt.plot(X, lin_reg_2.predict(poly_reg.fit_transform(X)), color = 'blue')
plt.title(), plt.xlabel(), plt.ylabel(), plt.show()
```

*** Instead of writing X_poly, We wrote the whole poly_reg.fit_trans(X) because our ~~poly_reg~~ ~~may~~ X_poly may test_set of X i.e X_test

Clear Results : [Polynomial, SVR, Decision TREE, Random Forest]

```
X_grid = np.arange(min(X), max(X), 0.1)
X_grid = ~~np.~~ X_grid.reshape((len(X_grid), 1))
plt.scatter(X, Y, color = 'red')
plt.plot(X_grid, lin_reg_2.predict(poly_reg.fit_transform(X_grid)),
         color = 'blue')
plt.show()
```

Visualising Test Set Results : [Linear, Multiple Linear]

```
plt.scatter(x_test, y_test, color = 'red')
plt.plot(x_train, regressor.predict(x_train), color = 'blue')
```

Visualising Training Set Results : [Linear, Multiple Linear]

```
plt.scatter(x_train, y_train, color = 'red')
plt.plot(x_train, regressor.predict(x_train), color = 'blue')
```

SVR Model : `sc_x = StandardScaler()`

```
from sklearn.svm import SVR
regressor = SVR(kernel = 'rbf')
regressor.fit(x, y)
y_pred = regressor.predict(sc_x.transform(6.5)) — X
```

Because the input to predict must be an array.

```
y_pred = regressor.predict(sc_x.transform(np.array(
                                [[6.5]])))
```

↓

Why 2nd bracket ?

If 1 bracket — vector

If 2 brackets — Array.

```
y_pred = sc_y.inverse_transform(reg.predict(sc_x.transform(...)))
```

↓

Because we are predicting Y value. Inverse Transform to inverse scaled value.

Decision Tree Regression:

```
from sklearn.tree import DecisionTreeRegressor
regressor = DecisionTreeRegressor(random_state = 0)
regressor.fit(X, y)
```

Random Forest Regression: Ensemble learning (or) Multiple Deep Forest Trees.

```
from sklearn.ensemble import RandomForestRegressor
regressor = RandomForestRegressor(n_estimator = 10, random_state=0)
regressor.fit(X, y)
y_pred = regressor.predict(6.5)
```

CLASSIFICATION

Logistic Regression :

```
from sklearn.linear_model import LogisticRegression
classifier = Logistic_Regression(random_state = 0)
classifier.fit(x_train, y_train)
y_pred = classifier.predict(x_test)
```

Confusion Matrix :

```
from sklearn.metrices import confusion_matrix
cm = confusion_matrix(y_test, y_pred)
>> cm
```

K-Nearest Neighbours :

```
from sklearn.neighbours import kNeighboursClassifier
classifier = kNeighboursClassifier(n_neighbours = 5,
                                   metric = 'minkowski', p = 2)
```

metric = 'minkowski' → To choose Euclidean Distance

```
classifier.fit(x_train, y_train)
y_pred = classifier.predict(x_test)
```

Support Vector Machine (SVM):

```
from sklearn.svm import SVC
classifier = SVC(kernel = 'linear', random_state = 0)
classifier.fit(X_train, y_train)
y_predict = classifier.predict(X_test)
```

Kernel SVM:

```
from sklearn.svm import SVC
classifier = SVC(kernel = 'rbf', random_state = 0)
classifier.fit(X_train, y_train)
y_pred = classifier.predict(X_test)
```

Naive_Bayes Theorem: $P(A/B) = \frac{P(B/A) * P(A)}{P(B)}$

*** See concept in the notebook (mandatory).

```
from sklearn.naive_bayes import GaussianNB
classifier = GaussianNB()
classifier.fit(X_train, y_train)
y_pred = classifier.predict(X_test)
```

Decision Tree Classification :

```
from sklearn.tree import DecisionTreeClassifier
classifier = DecisionTreeClassifier(criterion = 'entropy',
                                    random_state = 0)
classifier.fit(X_train, Y_train)
y_pred = classifier.predict(X_test)
```

Random Forest Classification :

```
from sklearn.ensemble import RandomForestClassifier
classifier = RandomForestClassifier(n_estimators = 10,
                                    criterion = 'entropy', random_state = 0)
classifier.fit(X_train, Y_train)
y_pred = classifier.predict(X_test)
```

CLUSTERING

k-Means Clustering : `from sklearn.cluster import kMeans`

```
#using elbow method
wcss = []
for i in range(1, 11):
    kmeans = kMeans(n_clusters = i, init = 'k-means++', max_iter = 300
                    n_init = 10, random_state = 0)
```

```
        kmeans.fit(x)                    ↗ method to calculate wcss.
        wcss.append(kmeans.inertia_)
plt.plot(range(1, 11), wcss)
plt.title( ), plt.xlabel( ), plt.ylabel( ), plt.show( )
```

Applying k-means to dataset

```
kmeans = kMeans(n_clusters = 5, init = 'k-means++', max_iter = 300,
                n_init = 10, random_state = 0)
y_kmeans = kmeans.fit_predict(x)
```

Hierarchical Clustering : Agglomerative Approach

```
import scipy.cluster.hierarchy as sch
dendogram = sch.dendogram(sch.linkage(dataset, method = 'ward'))
```

~~Agglomerative~~ → Algorithm

'ward' method in Algorithm

```
plt.plot( )
plt.xlabel( )
:
plt.show( )
```

Optimum clusters : 5

Fitting Hierarchical Clustering to the dataset

```
from sklearn, cluster import AgglomerativeClustering
hc = AgglomerativeClustering(n_clusters = 5, affinity = 'euclidean',
                             linkage = 'ward')
y_hc = hc.fit_predict(x)
```

Association Rule Learning

(6)

Apriori : SEE concept in notebook (Mandatory)

In Python, Apriori expects input as lists of lists.

```
dataset = pd.read_csv( ' ')
transactions = [ ]
for i in range(0, 7501): → convert to string
    transactions.append([str(dataset.values[i,j]
                        for j in range(0,20)])
```

↓ To make list of lists.

```
# training apriori with dataset
from apyori import apriori
rules = apriori(transactions, min_support = 0.003,
                min_confidence = 0.2, min_lift = 3, min_length = 2)
# visualising results
results = list(rules)
```