APPLIED ROOTS

AAIC Technologies Pvt Ltd

UNIVERSITY OF HYDERABAD

ADVANCED MACHINE LEARNING

Table of Contents

01. RANDOM FORESTS

RANDOM FORESTS

Random forests are a part of a subset of Machine Learning models called **Ensembles**. "Ensemble" simply means a group of independent entities working together to produce a desired effect. Ensemble methods basically combine multiple machine learning models into one compound model and use their combined predictive powers to make their final predictions. There are no limitations as to the type of models one can use within an ensemble system, but there are two ensemble models that have proven their effectiveness on a wide range of datasets for regression and classification, these are Random Forests and Gradient Boosted Trees, both of which uses decisions trees as their building blocks.

RANDOM FORESTS:

Random Forests ensembling is a technique used to overcome the basic problem of decision trees - that of overfitting the training data. A Random Forest is basically a collection of decision trees trained such that each of them **overfit** a different "version" of the training data. By aggregating the predictions of each of these overfit trees, we end with a model that gives much superior performance than any of these models individually on the original data.

Building Random Forests ensembles involves the following steps:

1. We first decide on the number of decision trees to be used in the ensemble. Say we decide to use **k** trees.
2. We then use a technique called **bootstrap sampling**. In this technique we create **k** versions of the original training data, by randomly sampling with replacement **n** samples from the original data, where **n** is the size of the original training data. Randomly sampling with replacement means that it is not required that each data point that is randomly sampled has to be unique (ie: The same data points can be sampled multiple times). Thus we end up with **k** versions of the data.
3. A unique decision tree is then fit on each of these **k** versions of the training data.
4. The algorithm of the decision tree is slightly modified in Random Forests. At each node/ branch while choosing the best splitting condition, the tree is allowed to use only a random subset of the original set of features, instead of all the features. This feature sampling procedure is repeated separately at each node for every tree and so each tree chooses its splitting condition using information from a different subset of the original set of features.

5. Each of the **k** trees are allowed to ovefit on the randomized versions of the data they are exposed to.
6. The predictions of each of these individual decision trees are then aggregated (majority of all the classes predicted by each tree for classification and average values predicted by each tree for regression).

As can be understood from the steps above, there are two levels of randomness infused into the tree building process - one at the data points sampling level and another at the features sampling level and hence the name "Random Forest", implying a randomized collection of decision trees.

RANDOM FORESTS USING SCIKIT LEARN:

Shown below is the scikit learn class that performs classification using Random Forests. Some of the more relevant parameters along with their default values are also displayed.

```
RandomForestClassifier(n_estimators = 100,
                       criterion = 'gini',
                       max_depth = None,
                   min_samples_split = 2,
                    min_samples_leaf = 1,
                       max_features = 'sqrt')
```

As can be seen from above, the parameters for the Random Forest classifier are similar to that of Decision Trees, except for two new parameters - **n_estimators** and **max_features**. The former parameter refers to the number of trees one wants to use in the ensemble and the latter specifies the size of the random feature subset selected when optimizing for the best splitting condition at each node. Three options are provided:**'sqrt'**, **'log2'** and **None**. If 'sqrt' is chosen, then the max_features = the rounded integer of the square root of the number of features and if 'log2' is chosen max_features = the rounded integer of the log2 of the number of features.

USING RANDOM FOREST ON BREAST CANCER DATA SET:

Since we have discussed the binary classification pipeline in quite some detail in the previous chapters, we will pick up from the "model training and evaluation" stage in the pipeline. We will be using the same set of machine learning Functions saved in a file "**aaic_mlfuncs.py**" .

We import the train and test sets as shown below:

```
import aaic_mlfuncs as ai
```

```
df_tr_set = pd.read_csv(data_path + 'df_tr_BC.csv')
df_ts_set = pd.read_csv(data_path + 'df_ts_BC.csv')

df_tr_set.shape, df_ts_set.shape
```

```
((438, 9), (114, 9))
```

```
X_train = df_tr_set.iloc[:, :-1].values
X_test  = df_ts_set.iloc[:, :-1].values

y_train = df_tr_set.iloc[:, -1].values
y_test  = df_ts_set.iloc[:, -1].values

X_train.shape, X_test.shape, y_train.shape, y_test.shape
```

```
((438, 8), (114, 8), (438,), (114,))
```

We Choose to perform grid search with K fold cross validation using the **RandomForestClassifier** class as shown below.

```
from sklearn.ensemble import RandomForestClassifier

model_class = RandomForestClassifier
parameter_grid = dict(n_estimators = range(50, 300, 50),
                      criterion = ['gini', 'entropy'],
                      max_depth = [None],
                  min_samples_split = range(2, 10),
                   min_samples_leaf = range(1, 10),
                       max_features = ['sqrt', 'log2'],
                       class_weight = ['balanced'],
                       random_state = [0])

z = ai.fn_kfoldcv_clf_binary(X_train, y_train, model_class, parameter_grid)

df_kfoldcv_gridsearch, dict0_model_instances = z
df_kfoldcv_gridsearch.describe()
```

```
100% (1440 of 1440) |####################| Elapsed Time: 0:21:29 Time:  0:21:29
```

	ts_mean_rec_1	ts_mean_prec_1	ts_mean_rec_0	ts_mean_prec_0	ts_mean_loss	ts_std_loss	ts_mean_acc	ts_std_rec_1	ts_std_rec_0
count	1440.000000	1440.000000	1440.000000	1440.000000	1440.000000	1440.000000	1440.000000	1440.000000	1440.000000
mean	0.963246	0.974076	0.953827	0.936685	0.127816	0.031943	0.959828	0.027756	0.032329
std	0.006495	0.003557	0.006489	0.010381	0.017288	0.024291	0.004773	0.005594	0.007501
min	0.950877	0.958105	0.924165	0.915776	0.101487	0.020935	0.948122	0.013129	0.015110
25%	0.957895	0.971774	0.949565	0.928571	0.117263	0.022895	0.957131	0.022739	0.023403
50%	0.964912	0.975011	0.955854	0.937107	0.124449	0.025291	0.959398	0.028220	0.031951
75%	0.968421	0.975583	0.955975	0.944744	0.133868	0.028602	0.963887	0.034379	0.038770
max	0.982456	0.979381	0.962264	0.968746	0.188144	0.126253	0.975179	0.038756	0.049415

We then filter the best models as shown below:

```
dff = df_kfoldcv_gridsearch
n = 5

df1 = dff.sort_values(by = 'ts_mean_rec_1', ascending = False)[:n]
df2 = dff.sort_values(by = 'ts_mean_prec_1', ascending = False)[:n]
df3 = dff.sort_values(by = 'ts_std_rec_1', ascending = True)[:n]
df4 = dff.sort_values(by = 'ts_mean_loss', ascending = True)[:n]
df5 = dff.sort_values(by = 'ts_std_loss', ascending = False)[:n]

df_filtered_cv = pd.concat([df1, df2, df3, df4, df5]).drop_duplicates()
df_filtered_cv = df_filtered_cv[df_filtered_cv.ts_mean_loss < 0.5]
df_filtered_cv = df_filtered_cv.sort_values(by = 'ts_mean_rec_1', ascending = False)[:5]
df_filtered_cv
```

	ts_mean_rec_1	ts_mean_prec_1	ts_mean_rec_0	ts_mean_prec_0	ts_mean_loss	ts_std_loss	ts_mean_acc	ts_std_rec_1	ts_std_rec_0
model_720	0.982456	0.979381	0.962264	0.968746	0.101961	0.021296	0.975179	0.013129	0.030811
model_757	0.982456	0.976120	0.955975	0.968485	0.102394	0.021523	0.972927	0.013129	0.038770
model_738	0.982456	0.976120	0.955975	0.968485	0.103390	0.022497	0.972927	0.013129	0.038770
model_1045	0.978947	0.979238	0.962264	0.962587	0.104937	0.023538	0.972927	0.014886	0.030811
model_433	0.978947	0.969030	0.943275	0.961874	0.101487	0.022806	0.966155	0.014886	0.030663

We can inspect the parameters of these chosen models as shown below:

```
models = [dict0_model_instances[i] for i in df_filtered_cv.index]

display(models)
```

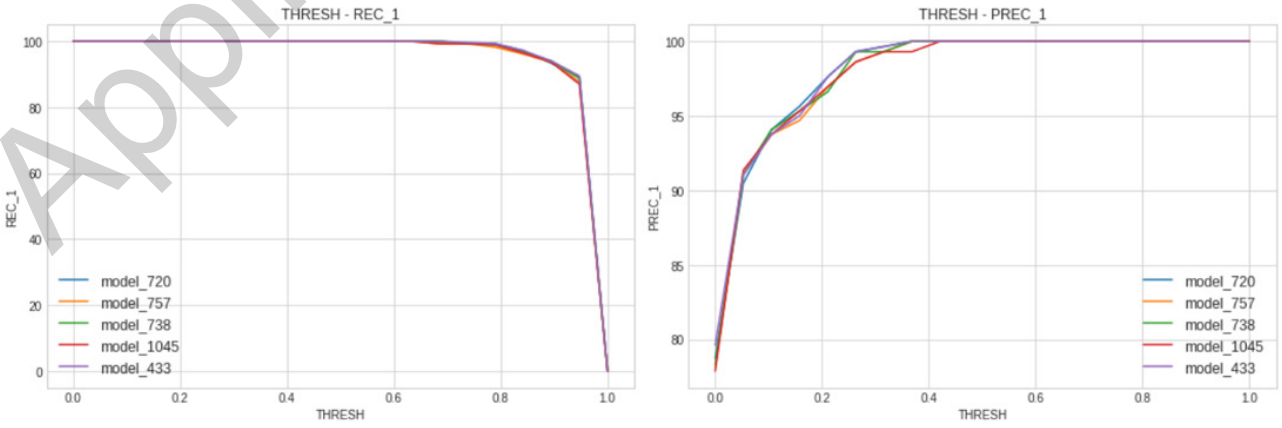
```
[RandomForestClassifier(bootstrap=True, ccp_alpha=0.0, class_weight='balanced',
                        criterion='entropy', max_depth=None, max_features='sqrt',
                        max_leaf_nodes=None, max_samples=None,
                        min_impurity_decrease=0.0, min_impurity_split=None,
                        min_samples_leaf=1, min_samples_split=2,
                        min_weight_fraction_leaf=0.0, n_estimators=150,
                        n_jobs=None, oob_score=False, random_state=0, verbose=0,
                        warm_start=False),
 RandomForestClassifier(bootstrap=True, ccp_alpha=0.0, class_weight='balanced',
                        criterion='entropy', max_depth=None, max_features='log2',
                        max_leaf_nodes=None, max_samples=None,
                        min_impurity_decrease=0.0, min_impurity_split=None,
                        min_samples_leaf=1, min_samples_split=4,
                        min_weight_fraction_leaf=0.0, n_estimators=150,
                        n_jobs=None, oob_score=False, random_state=0, verbose=0,
                        warm_start=False),
```

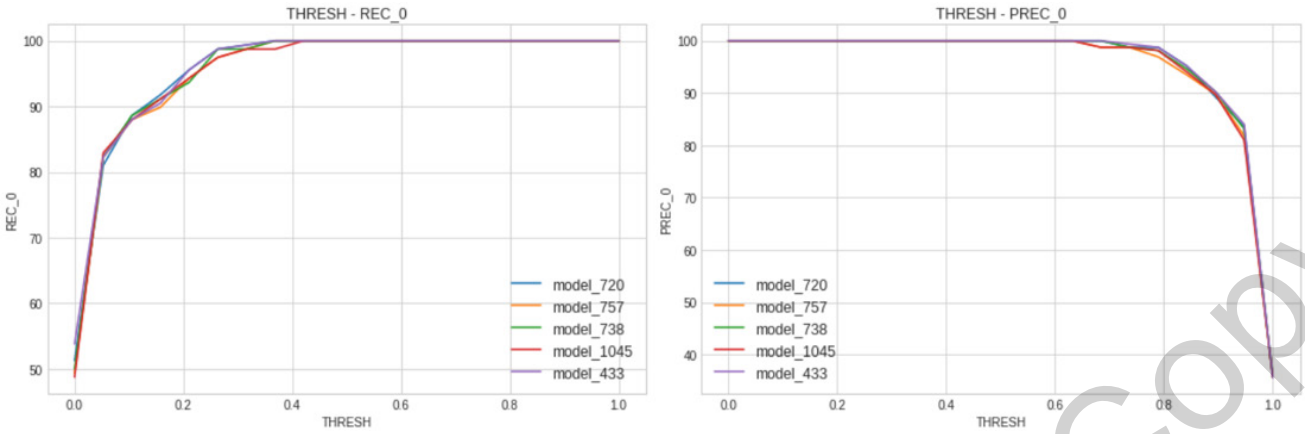
```
RandomForestClassifier(bootstrap=True, ccp_alpha=0.0, class_weight='balanced',
                       criterion='entropy', max_depth=None, max_features='sqrt',
                       max_leaf_nodes=None, max_samples=None,
                       min_impurity_decrease=0.0, min_impurity_split=None,
                       min_samples_leaf=1, min_samples_split=3,
                       min_weight_fraction_leaf=0.0, n_estimators=150,
                       n_jobs=None, oob_score=False, random_state=0, verbose=0,
                       warm_start=False),
RandomForestClassifier(bootstrap=True, ccp_alpha=0.0, class_weight='balanced',
                       criterion='entropy', max_depth=None, max_features='log2',
                       max_leaf_nodes=None, max_samples=None,
                       min_impurity_decrease=0.0, min_impurity_split=None,
                       min_samples_leaf=1, min_samples_split=4,
                       min_weight_fraction_leaf=0.0, n_estimators=200,
                       n_jobs=None, oob_score=False, random_state=0, verbose=0,
                       warm_start=False),
RandomForestClassifier(bootstrap=True, ccp_alpha=0.0, class_weight='balanced',
                       criterion='entropy', max_depth=None, max_features='log2',
                       max_leaf_nodes=None, max_samples=None,
                       min_impurity_decrease=0.0, min_impurity_split=None,
                       min_samples_leaf=1, min_samples_split=2,
                       min_weight_fraction_leaf=0.0, n_estimators=100,
                       n_jobs=None, oob_score=False, random_state=0, verbose=0,
                       warm_start=False)]
```

As can be seen from the above model instances, the **entropy criteria** yields the best results. For the **max_features** parameter, both 'sqrt' and 'log2' are used by these top performing models.

We then plot the precision - recall curves for the above selected models as shown below:

```
df_tr_standard, df_ts_standard, scaler = ai.fn_standardize_df(df_tr_set, df_ts_set)

list0_model_names = list(df_filtered_cv.index)
X_train, y_train = df_tr_standard.iloc[:, :-1].values, df_tr_standard.iloc[:, -1].values
list0_models = [dict0_model_instances[i].fit(X_train, y_train) for i in list0_model_names]

list0_thresholds = np.linspace(0, 1, 20)
legend = list0_model_names

ai.fn_performance_models_data(list0_models, df_tr_standard, list0_thresholds, legend)
```





Assuming that we want to optimize for precision, we see that **model_433** thresholded at around 0.6, will most likely give best precision performance, while at the same time keeping all other metrics as high as possible. We then train the model_433 over the entire train set and check its performance across the train and test sets as shown below:

```
df_Xy_ = df_tr_standard
model_ = dict0_model_instances['model_433'].fit(X_train, y_train)
thresh = 0.6

ai.fn_test_model_binary_clf(df_Xy_, model_, threshold_class_1 = thresh)
```

```
--------------------
LOGLOSS : 0.026
ACCURACY: 100.0
--------------------
```

	prec	rec
class_0	100.0	100.0
class_1	100.0	100.0

```
df_Xy_ = df_ts_standard

ai.fn_test_model_binary_clf(df_Xy_, model_, threshold_class_1 = thresh)
```

```
--------------------
LOGLOSS : 0.0849
ACCURACY: 98.246
--------------------
```

	prec	rec
class_0	97.619	97.619
class_1	98.611	98.611

From the performances indicated in the outputs above we see that the model gives 100% accuracy on the train set and it generalizes this performance to the test set, unlike in the case of using a single decision tree.

ADVANTAGES AND DISADVANTAGES OF RANDOM FORESTS:

Random forests share all the benefits of decision trees, while at the same time make up for all their limitations. They are simple to understand and do not require extensive hyper parameter tuning to get good performance. Since all the trees in the random forests model are grown independently, they are easily parallelizable making them suited for handling large datasets.

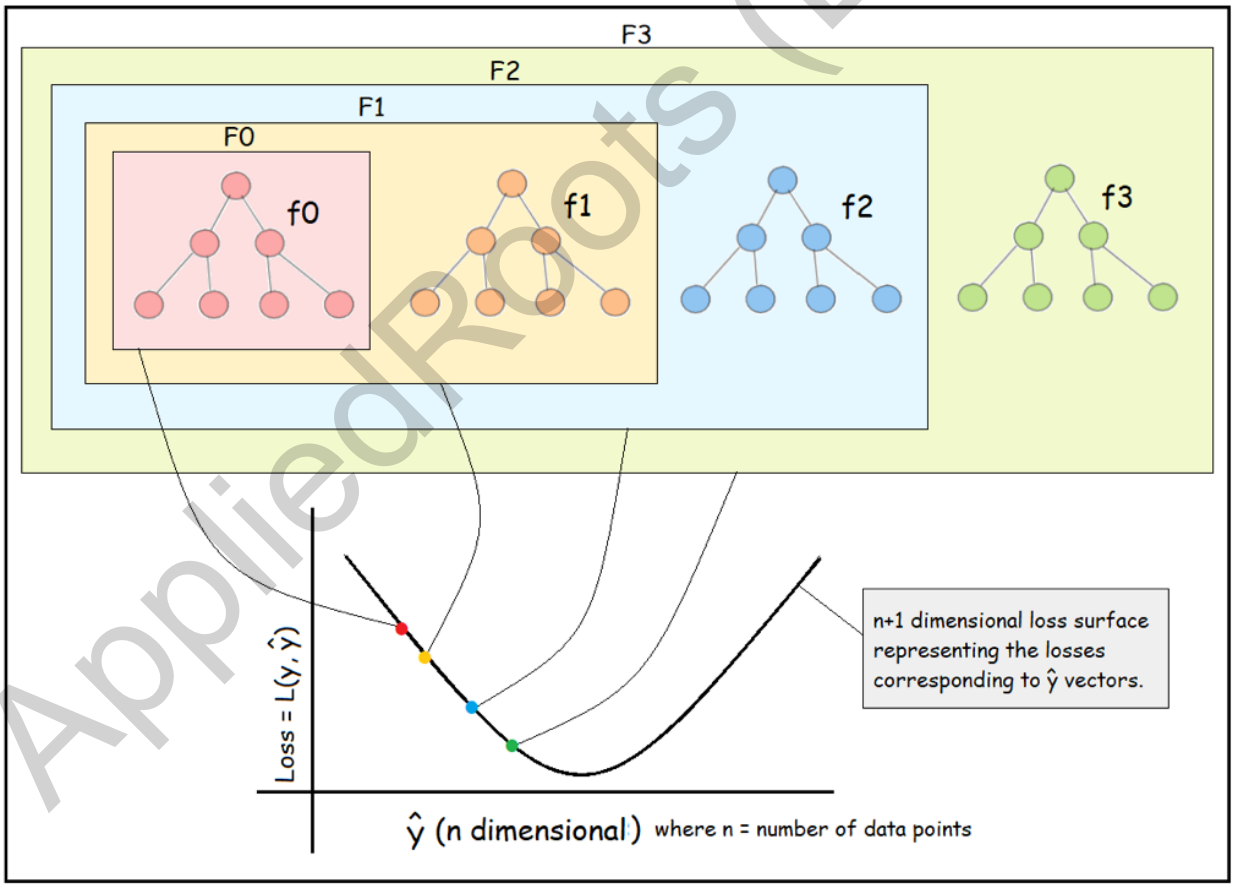
The down side of random forests is that they lose the visual interpretability of single tree models and are slower to train than linear models.

02. GRADIENT BOOSTED DECISION TREES

GRADIENT BOOSTED DECISION TREES

Gradient Boosted Decision Trees (GBDT) is another tree based ensemble method that uses the collective predictive power of multiple decision trees to create a single more powerful model. It differs from Random Forests in the sense that it does not create multiple independent trees but instead GBDTs create sequential models, where each tree tries to improve the prediction of the previous model.

The main idea behind gradient boosting is that we use a sequence of weak learners (shallow trees). Let's explain GBDTs using the case of regression. Consider the image shown below. The loss function depicted in the image is a simplified representation of a **n+1** dimensional regression loss space, where the horizontal axis represents various **n** dimensional prediction vectors (ie: The vector containing all predictions corresponding to each data point in **X**, ie **n** = number of data points in **X**).



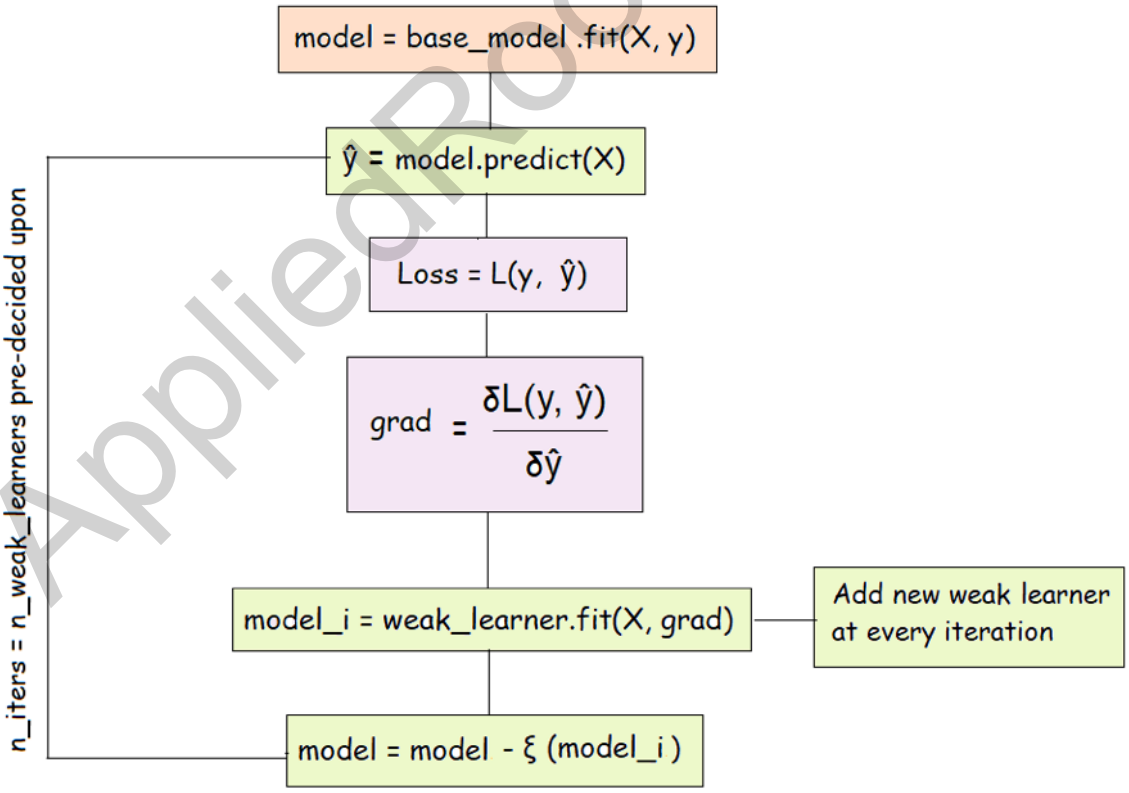
1. The first weak learner **f0** (tree represented as a function **f0**) is trained on the data **X** and labels **y**. This model **f0** is called the "base model". It is then made to predict on **X**, thus getting the prediction vector $\hat{y} = f0(X)$. Let's say it produces a squared loss = square$(y - \hat{y})$ = **L0**

2. We now compute the **gradient** of the squared loss with respect to **ŷ** (ie: -2(y - ŷ)) at that particular **ŷ** corresponding to **f0**. Let's call this vector **grad_y**.
3. We then train another weak learner **f1** on the data **X** and use **grad_y** as its target variable. In other words **f1** is trained to predict the gradient at that point on the loss surface, that corresponds to the prediction vector **ŷ** of **f0**. Lets call the gradient predicted by **f1** as **grad_ŷ**.
4. Now **ŷ - ξ(grad_ŷ)** = **ŷ_new** will be a better prediction that **ŷ** as the loss corresponding to it will be lesser than **L0**. In other words the combined effects of weak learners **f0** and **f1** results in a new prediction vector **ŷ_new** that produces lesser loss. Here ξ is the learning rate. We call this combined model **F1**. Hence **F1 = f0 + f1**
5. Next we apply steps 2, 3 and 4 on **F1** and recursively do so for every successive model combination that arises. So **F2** will be = **F1 + f2, F3** will be = **F2+ f3** and so on. In general we have:

$$F_m = F_0(X) + \sum_{i=1}^{m} f_i(X)$$

As the value of **i** (the number of shallow learners) in the above equation increases, model **Fm** becomes more and more better until we reach the minima of the loss surface.

The image below expresses the above steps in the form of a flow chart.



GBDTS FOR CLASSIFICATION:

In the case of classification the process for GBDTs is similar, except that:

1. The base learner performs classification and predicts the probabilities for the classes in the dataset. Cross entropy is used as the loss function.
1. All subsequent weak learners (ie: f1, f2, f3, etc) perform regression and try to predict the gradient of the loss surface with respect to the previous compound model's prediction.
1. All subsequent compound models (ie: F1, F2, F3, etc) perform classification and predict class probabilities.

GBDTS USING SCIKIT LEARN:

Shown below is the scikit learn class that performs classification using GBDTs. Some of the more relevant parameters along with their default values are also displayed.

```
GradientBoostingClassifier(learning_rate = 0.1,
                           n_estimators = 100,
                           min_samples_split = 2,
                           min_samples_leaf = 1,
                           max_depth = 3,
                           max_features = 'sqrt')
```

As can be seen from above, the parameters for the GBDT classifier are similar to that of Random Forests, except for the learning_rate parameter.

USING GBDTs ON BREAST CANCER DATA SET:

Since we have discussed the binary classification pipeline in quite some detail in the previous chapters, we will pick up from the "model training and evaluation" stage in the pipeline. We will be using the same set of machine learning Functions saved in a file "**aaic_mlfuncs.py**" .

We import the train and test sets as shown below:

```
import aaic_mlfuncs as ai
```

```
df_tr_set = pd.read_csv(data_path + 'df_tr_BC.csv')
df_ts_set = pd.read_csv(data_path + 'df_ts_BC.csv')

df_tr_set.shape, df_ts_set.shape
```

```
((438, 9), (114, 9))
```

```
X_train = df_tr_set.iloc[:, :-1].values
X_test  = df_ts_set.iloc[:, :-1].values

y_train = df_tr_set.iloc[:, -1].values
y_test  = df_ts_set.iloc[:, -1].values

X_train.shape, X_test.shape, y_train.shape, y_test.shape
```

```
((438, 8), (114, 8), (438,), (114,))
```

We Choose to perform grid search with K fold cross validation using the **GradientBoostingClassifier** class as shown below.

```
from sklearn.ensemble import GradientBoostingClassifier

model_class = GradientBoostingClassifier

parameter_grid = dict(learning_rate = [0.0001, 0.001, 0.01, 0.1],
                      n_estimators = [100, 200, 300, 400],
                      min_samples_split = [2],
                      min_samples_leaf = [1],
                      max_depth = [3, 4, 5, 6, 7],
                      max_features = ['sqrt', 'log2'])

z = ai.fn_kfoldcv_clf_binary(X_train, y_train, model_class, parameter_grid)

df_kfoldcv_gridsearch, dict0_model_instances = z
df_kfoldcv_gridsearch.describe()
```

```
100% (160 of 160) |######################| Elapsed Time: 0:02:15 Time:  0:02:15
```

	ts_mean_rec_1	ts_mean_prec_1	ts_mean_rec_0	ts_mean_prec_0	ts_mean_loss	ts_std_loss	ts_mean_acc	ts_std_rec_1	ts_std_rec_0
count	160.000000	160.000000	1.600000e+02	160.000000	160.000000	160.000000	160.000000	160.000000	1.600000e+02
mean	0.989232	0.832972	5.608423e-01	0.979577	0.356966	0.020952	0.836427	0.007240	2.329943e-02
std	0.009949	0.149288	4.388261e-01	0.018298	0.216661	0.019924	0.150838	0.006423	2.027197e-02
min	0.968421	0.643347	1.898887e-08	0.943862	0.107948	0.001181	0.643347	0.000000	1.710466e-10
25%	0.978947	0.643347	1.898887e-08	0.961490	0.143905	0.002483	0.643347	0.000000	1.710466e-10
50%	0.989474	0.924843	8.543179e-01	0.976828	0.344131	0.012062	0.941281	0.008595	3.050493e-02
75%	1.000000	0.962191	9.304548e-01	1.000000	0.589655	0.035083	0.961620	0.013129	3.928206e-02
max	1.000000	0.975945	9.559748e-01	1.000000	0.643579	0.064971	0.970659	0.022739	7.357073e-02

We then filter the best models as shown below:

```
dff = df_kfoldcv_gridsearch
n = 5

df1 = dff.sort_values(by = 'ts_mean_rec_1', ascending = False)[:n]
df2 = dff.sort_values(by = 'ts_mean_prec_1', ascending = False)[:n]
df3 = dff.sort_values(by = 'ts_std_rec_1', ascending = True)[:n]
df4 = dff.sort_values(by = 'ts_mean_loss', ascending = True)[:n]
df5 = dff.sort_values(by = 'ts_std_loss', ascending = False)[:n]

df_filtered_cv = pd.concat([df1, df2, df3, df4, df5]).drop_duplicates()
df_filtered_cv = df_filtered_cv[df_filtered_cv.ts_mean_loss < 0.5]
df_filtered_cv = df_filtered_cv.sort_values(by = 'ts_mean_rec_1', ascending = False)[:5]
df_filtered_cv
```

	ts_mean_rec_1	ts_mean_prec_1	ts_mean_rec_0	ts_mean_prec_0	ts_mean_loss	ts_std_loss	ts_mean_acc	ts_std_rec_1	ts_std_rec_0
model_114	0.985965	0.962508	0.930455	0.973323	0.110370	0.034596	0.966140	0.009924	0.031931
model_112	0.982456	0.965833	0.936865	0.967527	0.107948	0.030207	0.966140	0.013129	0.035408
model_113	0.982456	0.968927	0.943154	0.967572	0.109325	0.031160	0.968407	0.013129	0.026514
model_136	0.982456	0.969341	0.943275	0.967876	0.150025	0.060713	0.968407	0.013129	0.040648
model_133	0.978947	0.975654	0.955854	0.962251	0.136086	0.049792	0.970659	0.014886	0.023403

We can inspect the parameters of these chosen models as shown below:

```
models = [dict0_model_instances[i] for i in df_filtered_cv.index]

display(models)
```

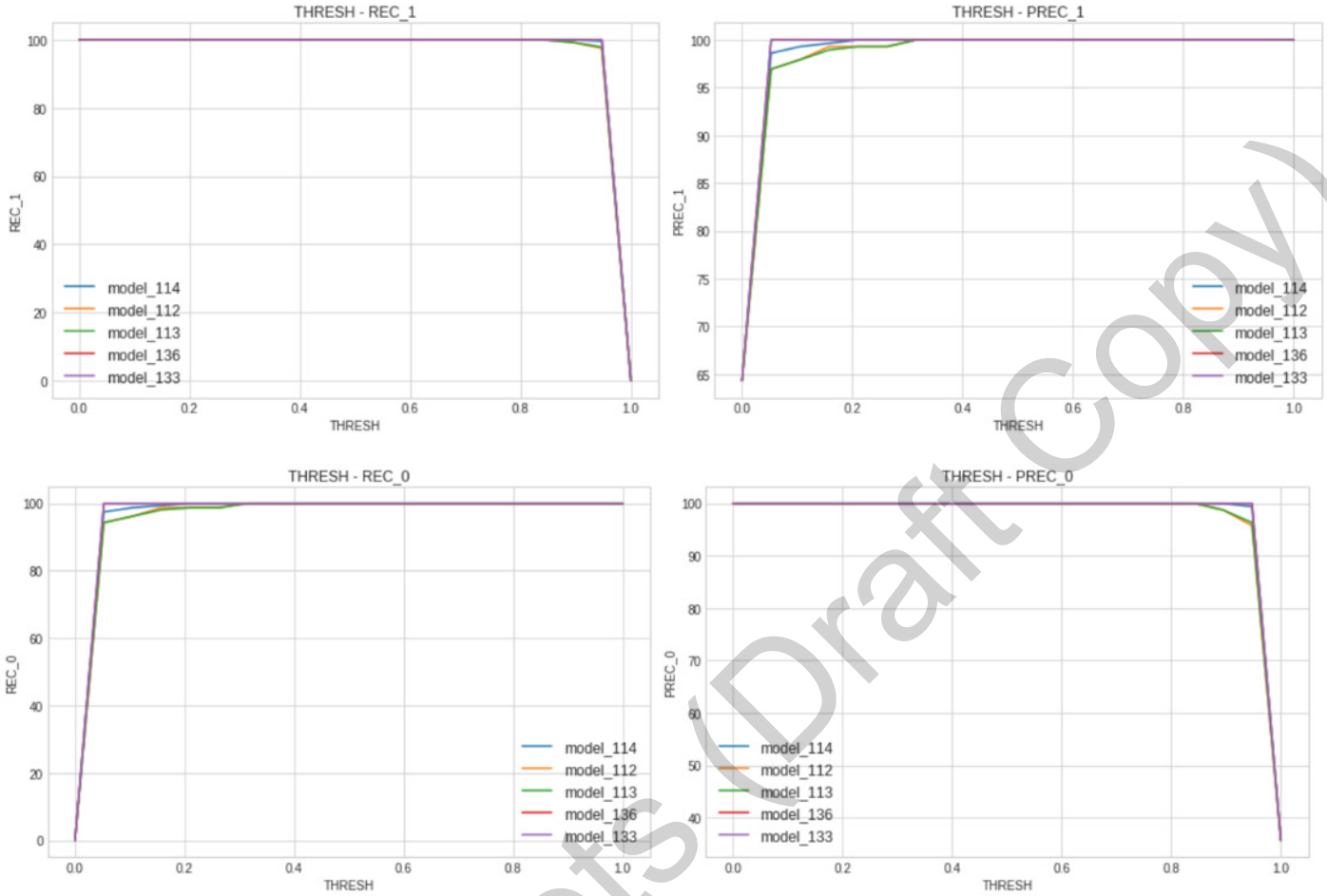
```
[GradientBoostingClassifier(ccp_alpha=0.0, criterion='friedman_mse', init=None,
                            learning_rate=0.01, loss='deviance', max_depth=5,
                            max_features='sqrt', max_leaf_nodes=None,
                            min_impurity_decrease=0.0, min_impurity_split=None,
                            min_samples_leaf=1, min_samples_split=2,
                            min_weight_fraction_leaf=0.0, n_estimators=400,
                            n_iter_no_change=None, presort='deprecated',
                            random_state=None, subsample=1.0, tol=0.0001,
                            validation_fraction=0.1, verbose=0,
                            warm_start=False),
 GradientBoostingClassifier(ccp_alpha=0.0, criterion='friedman_mse', init=None,
                            learning_rate=0.01, loss='deviance', max_depth=4,
                            max_features='sqrt', max_leaf_nodes=None,
                            min_impurity_decrease=0.0, min_impurity_split=None,
                            min_samples_leaf=1, min_samples_split=2,
                            min_weight_fraction_leaf=0.0, n_estimators=400,
                            n_iter_no_change=None, presort='deprecated',
                            random_state=None, subsample=1.0, tol=0.0001,
                            validation_fraction=0.1, verbose=0,
                            warm_start=False),
```

```
GradientBoostingClassifier(ccp_alpha=0.0, criterion='friedman_mse', init=None,
                           learning_rate=0.01, loss='deviance', max_depth=4,
                           max_features='log2', max_leaf_nodes=None,
                           min_impurity_decrease=0.0, min_impurity_split=None,
                           min_samples_leaf=1, min_samples_split=2,
                           min_weight_fraction_leaf=0.0, n_estimators=400,
                           n_iter_no_change=None, presort='deprecated',
                           random_state=None, subsample=1.0, tol=0.0001,
                           validation_fraction=0.1, verbose=0,
                           warm_start=False),
GradientBoostingClassifier(ccp_alpha=0.0, criterion='friedman_mse', init=None,
                           learning_rate=0.1, loss='deviance', max_depth=6,
                           max_features='sqrt', max_leaf_nodes=None,
                           min_impurity_decrease=0.0, min_impurity_split=None,
                           min_samples_leaf=1, min_samples_split=2,
                           min_weight_fraction_leaf=0.0, n_estimators=200,
                           n_iter_no_change=None, presort='deprecated',
                           random_state=None, subsample=1.0, tol=0.0001,
                           validation_fraction=0.1, verbose=0,
                           warm_start=False),
GradientBoostingClassifier(ccp_alpha=0.0, criterion='friedman_mse', init=None,
                           learning_rate=0.1, loss='deviance', max_depth=4,
                           max_features='log2', max_leaf_nodes=None,
                           min_impurity_decrease=0.0, min_impurity_split=None,
                           min_samples_leaf=1, min_samples_split=2,
                           min_weight_fraction_leaf=0.0, n_estimators=200,
                           n_iter_no_change=None, presort='deprecated',
                           random_state=None, subsample=1.0, tol=0.0001,
                           validation_fraction=0.1, verbose=0,
                           warm_start=False)]
```

As can be seen from the above model instances, the models with more estimators yield the best results. For the l**earning_rate** and **depth** parameters, values of 0.01 and 5 turns out to be the best.

We then plot the precision - recall curves for the above selected models as shown below:

```
df_tr_standard, df_ts_standard, scaler = ai.fn_standardize_df(df_tr_set, df_ts_set)

list0_model_names = list(df_filtered_cv.index)
X_train, y_train = df_tr_standard.iloc[:, :-1].values, df_tr_standard.iloc[:, -1].values
list0_models = [dict0_model_instances[i].fit(X_train, y_train) for i in list0_model_names]

list0_thresholds = np.linspace(0, 1, 20)
legend = list0_model_names

ai.fn_performance_models_data(list0_models, df_tr_standard, list0_thresholds, legend)
```



Assuming that we want to optimize for precision, we see that **model_114** thresholded at around 0.7, will most likely give best precision performance, while at the same time keeping all other metrics as high as possible. We then train the model_433 over the entire train set and check its performance across the train and test sets as shown below:

```
df_Xy_ = df_tr_standard
model_ = dict0_model_instances['model_114'].fit(X_train, y_train)
thresh = 0.7
ai.fn_test_model_binary_clf(df_Xy_, model_, threshold_class_1 = thresh)
```

```
--------------------
LOGLOSS : 0.0119
ACCURACY: 100.0
--------------------
```

	prec	rec
class_0	100.0	100.0
class_1	100.0	100.0

```
df_Xy_ = df_ts_standard

ai.fn_test_model_binary_clf(df_Xy_, model_, threshold_class_1 = thresh)
```

```
--------------------
LOGLOSS : 0.0704
ACCURACY: 98.246
--------------------
```

	prec	rec
class_0	97.619	97.619
class_1	98.611	98.611

From the performances indicated in the outputs above we see that the model gives 100% accuracy on the train set and it generalizes well to the test set.

GBDT LOSS FUNCTION:

It should be noted that the loss functions used in GBDTs are not directly connected to the model in any way. The gradient of the loss function is computed only with respect to $\hat{\mathbf{y}}$, where $\hat{\mathbf{y}}$ is not further deconstructed as a function of the model being used. In other words, the loss function gradient is not directly connected to any of the model's internal parameters.

ADVANTAGES AND LIMITATIONS OF GBDTS:

GBDTs provide cutting edge performance and at present they along with Deep Learning models are the most widely used in cases where very high performance is expected. The performance and generalization of the model improves with the number of estimators used.

As with the Random Forest model, and even more so for GBDTs, interpretability is lost as the model complexity increases with each estimator used. Since in GBDTs every classifier is obliged to fix the errors in the predecessors, it could be sensitive to outliers if the first tree is fitted sub optimally with noisy data.

03. MODEL CALIBRATION

MODEL CALIBRATION

In the previous chapters on binary classification we trained our models and then optimized them for metrics such as accuracy, precision and recall by finding the best prediction threshold. We chose the best prediction threshold by inspecting the precision and recall curves of both the positive and negative classes and then choosing that threshold that maximised the metric we desire (while compromising the other metrics as less as possible). This approach is most often used when we are interested in creating a model that predicts the classes as best as possible.

Apart from predicting correct **classes**, there may be situations where the **confidence** that the model has on its prediction (the **probability** of the predicted class) is more important. Consider the case of breast cancer classification. Say that the model predicts that a patient has cancer, but when we look into the **confidence** that the model has on that particular prediction and see that it is 50.1%, it produces a different perspective to the situation.

So in cases like the one described above, we need the predicted probabilities of the model to get a better perspective. This is where the technique of **model calibration** comes in.

RELATION BETWEEN CONFIDENCE (PREDICTED PROBABILITY) AND ACCURACY:

Consider the following two classifiers:

Model_1: 80% Accuracy and 79% confidence in each prediction.

Model_2: 80% accuracy and 90% confidence in each prediction.

Model_1 is better since its confidence is validated by the model's accuracy. In other words, given an abstract case where the model has a confidence of 80% on all of its predictions, it implicitly means that the model will predict the correct class (positive) 80% of time.

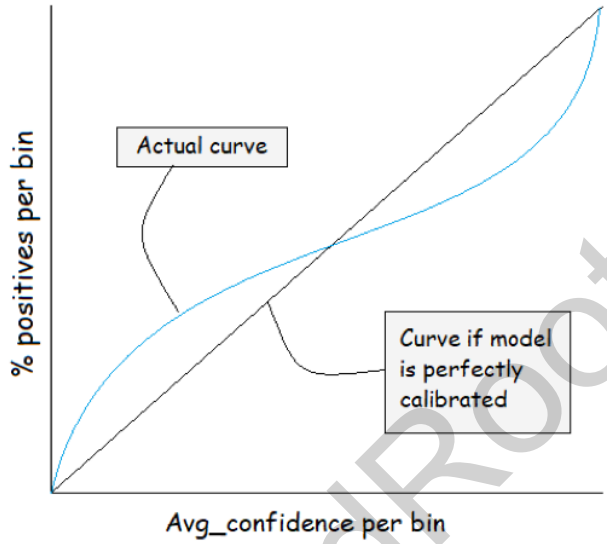
A well calibrated binary classifier should classify the samples such that among the samples to which it gave a predict_proba value close to 0.8, approximately 80% actually belong to the positive class.

Suppose we sort the predicted probability values (ie: confidence values) of the predictions of a model and bin them into five bins and create a table as shown below, then the values of the last column (% positives per bin) and the values of the third column (avg_pred_proba of bin) should be similar or close to each other.

Bin_num	Bin_range	avg_pred_proba	n_positives	n_preds	% positives per bin
1	81 - 100	P1	n_pos_1	total_1	n_pos_1 / total_1
2	61 - 80	P2	n_pos_2	total_2	n_pos_2 / total_2
3	41 - 60	P3	n_pos_3	total_3	n_pos_3 / total_3
4	21 - 40	P4	n_pos_4	total_4	n_pos_4 / total_4
5	0 - 20	P5	n_pos_5	total_5	n_pos_5 / total_5

CALIBRATION PLOT (RELIABILITY CURVE):

A calibration plot is used to visualize how reliable the probabilities predicted by the model are. It basically is a plot of the confidence per bin versus the fraction of positives per bin (columns 2 Vs the last column of the table above). The curve that results is called the reliability curve. The more the reliability curve of a model deviates from the curve associated with perfect calibration, the lesser the reliability of the probabilities it predicts.



TYPICAL CALIBRATION PERFORMANCE OF DIFFERENT MODELS:

Different models have different reliability with respect to the probabilities they predict. In general Logistic regression models are inherently well calibrated, since we use the log loss to optimize its performance (log loss is based on the probabilities predicted by the model).

Gaussian Naive Bayes models tend to push probabilities towards 0 or 1. This is basically because it makes the naive assumption that features of the data set are conditionally independent given the class, which may not be the case.

Ensemble models like Random Forests and Gradient boosted decision Trees show the opposite behavior. These models aggregate predictions from multiple learners and so have difficulty making predictions near 0 and 1 because the variance among the different learners will bias predictions that should be near zero or one away from these values (All learners should have equally high confidence for high probability predictions by the ensemble as a whole, which is rare).

Support Vector Machines show a similar behaviour to that of ensemble models, they tend to push their probability estimates away from 0 & 1. This is because they optimize for larger margins and are mainly focused on the dataset's support vector points (ie: Points from opposite classes, closest to the separating plane. Thus it comprises confidence with respect to points away from the separating plane, which it actually should be very confident about.

In general it is a good practice to calibrate our models when we are applying our models to situations where the probabilities it predicts (confidence) is an important criteria.

MODEL CALIBRATION TECHNIQUES:

There two main techniques used to calibrate the probability prediction of models:

1. Platt Scaling

2. Isotonic regression

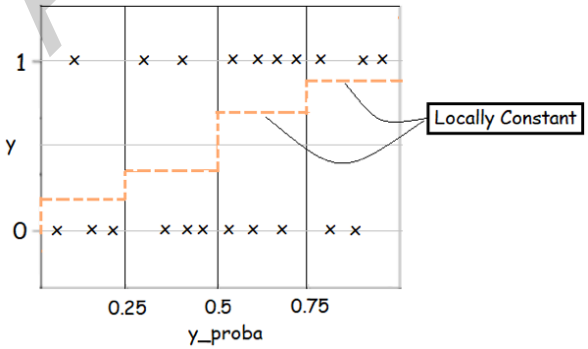
PLATT SCALING:

Platt scaling in essence involves using a logistic regression model as shown below. In essence it involves fitting a logistic regression model to the predicted probabilities of the uncalibrated model (y_proba_train) and the actual corresponding class labels (y_train), where the latter serves as the labels.

```
from sklearn.linear_models import LogisticRegression

calibration_model = LogisticRegression().fit(y_proba_train, y_train)
calibrated_predictions = calibration_model.predict(y_proba_test)
```

ISOTONIC REGRESSION:

Isotonic regression in essence involves fitting a **monotonous piecewise constant function** to transform the uncalibrated outputs of the original model. A function is said to be piecewise constant if it is locally constant. This in essence means that we fit a isotonic linear regression model, where **X** is the probabilities predicted for each datapoint and **y** the actual labels. The details of isotonic regression are beyond the scope of this book, but it produces the kind of prediction shown in the image below. This reduces the loss (or difference) between the probabilities predicted and the actual class and in effect calibrates the model's probabilities.



The Isotonic regression technique has a tendency to overfit and should be avoided when dealing with small datasets.

In general Platt Scaling calibrates the probability predictions quite well without the risk of overfitting and is more often due to its simplicity.

MODEL CALIBRATION USING SCIKIT LEARN:

Shown below is the scikit learn class that performs probability calibration. Some of the more relevant parameters along with their preferred values are also displayed.

```
CalibratedClassifierCV(base_estimator = base_model,
                       method ='sigmoid',
                          cv ='prefit')
```

BASE_ESTIMATOR: This parameter refers to the original model whose predicted probabilities need to be calibrated. Here we have the following choices:

1. Use an untrained model class instance. For this the cv parameter has to be set to some positive integer, representing the number of cross validation splits to use while training the combination of the original model and the calibrating model.
2. Use a trained model class instance. For this the cv parameter has to be set to **"prefit"** as shown above.

METHOD: This parameter refers to the method used for calibration. Two options are available **'sigmoid'**, which means Platt scaling (default setting) and **'isotonic'.**

USING CALIBRATION ON RANDOM FOREST FOR BREAST CANCER DATA SET:

For the sake of demonstration, we will use the Random Forest model with default settings. We will pick up from the "model training and evaluation" stage in the pipeline. We will be using the same set of machine learning functions saved in a file **"aaic_mlfuncs.py"** .

We import the train and test sets as shown below:

```
import aaic_mlfuncs as ai
```

```
df_tr_set = pd.read_csv(data_path + 'df_tr_BC.csv')
df_ts_set = pd.read_csv(data_path + 'df_ts_BC.csv')

df_tr_set.shape, df_ts_set.shape
```

```
((438, 9), (114, 9))
```

We then fit a Random Forest classifier (base estimator) to the train set and check its performance on the train and test sets as shown below.

```
from sklearn.ensemble import RandomForestClassifier

df_tr_standard, df_ts_standard, scaler = ai.fn_standardize_df(df_tr_set, df_ts_set)

X_train = df_tr_standard.iloc[:, :-1].values
y_train = df_tr_standard.iloc[:, -1].values

model_ = RandomForestClassifier(random_state = 0).fit(X_train, y_train)

df_Xy_ = df_tr_standard

ai.fn_test_model_binary_clf(df_Xy_, model_)
```

```
--------------------
LOGLOSS : 0.0289
ACCURACY: 100.0
--------------------
```

	prec	rec
class_0	100.0	100.0
class_1	100.0	100.0

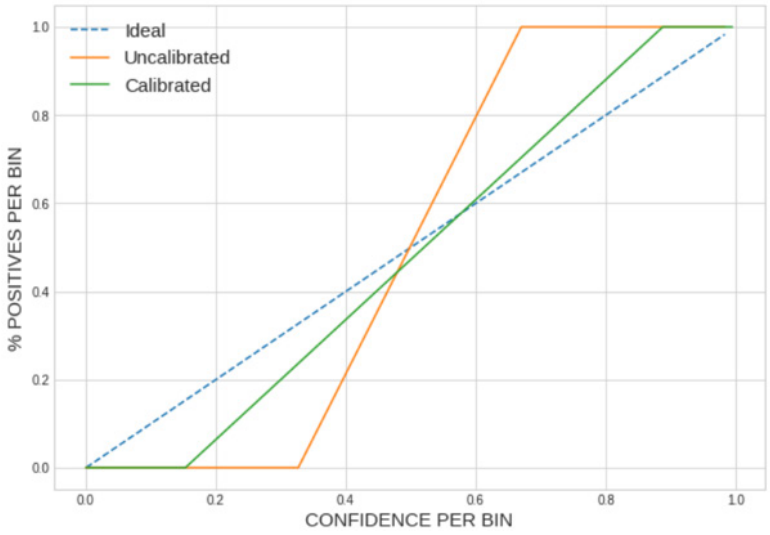
```
df_Xy_ = df_ts_standard

ai.fn_test_model_binary_clf(df_Xy_, model_)
```

```
--------------------
LOGLOSS : 0.0818
ACCURACY: 97.368
--------------------
```

	prec	rec
class_0	97.561	95.238
class_1	97.260	98.611

We then perform model calibration as shown below:

```
from sklearn.calibration import CalibratedClassifierCV

base_model = model_
calibrated_clf_instance = CalibratedClassifierCV(base_estimator = base_model, cv='prefit')
calibrated_clf = calibrated_clf_instance.fit(X_train, y_train)

original_probas = base_model.predict_proba(X_train)[:, 1]
calibrated_probas = calibrated_clf.predict_proba(X_train)[:, 1]

y_, y_proba, y_calibrated_proba = y_train, original_probas, calibrated_probas

ai.fn_calibration_plot(y_, y_proba, y_calibrated_proba)
```



As can be seen from the Calibration plot above, the calibration curve of the calibrated model deviates lesser from the ideal curve.

We then check the performance of the calibrated model as shown below:

```
df_Xy_ = df_tr_standard
model_ = calibrated_clf

ai.fn_test_model_binary_clf(df_Xy_, model_)
```

```
--------------------
LOGLOSS : 0.0058
ACCURACY: 100.0
--------------------
```

	prec	rec
class_0	100.0	100.0
class_1	100.0	100.0

```
df_Xy_ = df_ts_standard

ai.fn_test_model_binary_clf(df_Xy_, model_)
```

```
--------------------
LOGLOSS : 0.0602
ACCURACY: 97.368
--------------------
```

	prec	rec
class_0	97.561	95.238
class_1	97.260	98.611

As can be seen from the performance indicated above, there is a drastic reduction in the log loss with respect to the train set and a noticeable reduction of the same with respect to the test set. The count performances of the model - Accuracy, precision and recall may not change (much) post calibration.

It should be noted that calibration is not always necessary. It is only used when the confidence predictions of the model are important with respect to the task at hand. Calibration need not necessarily mean better class predictions, as was just mentioned earlier. Calibration only means that the probabilities/confidence behind the predictions have been "corrected" to a certain degree.

04. INTRO TO NLP (SENTIMENT CLASSIFICATION)

INTRO TO NLP (SENTIMENT CLASSIFICATION)

In this chapter we will discuss the concept of sentiment classification, by using a data set that basically contains text documents that represent product reviews given by customers, along with their corresponding product ratings (1 to 5 stars). This will provide a basis for understanding how non numerical data like Natural Language can be Processed and used to train Machine Learning models.

The study of natural language processing has been around for more than fifty years and grew out of the field of linguistics with the rise of computers. It generally deals with building computational tools that do useful things with language like sentiment classification, machine translation, document summarization, question-answering to mention a few.

Consider the case of a dataset consisting of product reviews, given by customers. The basic technical parlance for text processing is as follows:

- The collection of all the text documents (ie: Product reviews) would be called a **Corpus**.
- Each text document will simply be called a **Document**.
- The set of all the words contained in the Corpus is called its **Vocabulary**.

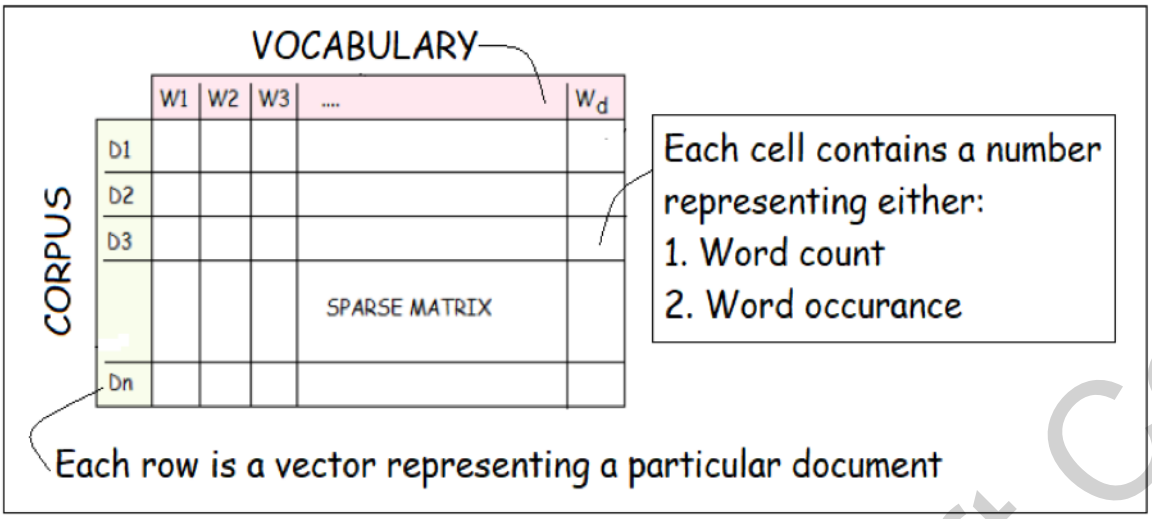
Here, we wish to represent each document as a vector but due to the nature of textual data, it is not as obvious how to vectorize a document or create features for a corpus. Over the years the field of Natural Language processing **(NLP)** has come up with many tried and tested techniques to featurize text data and newer impressive techniques keep emerging. In this chapter, we will focus on 3 widely used text vectorization techniques:

1. Bag of words (**BOWs**).
2. Term Frequency inverse document frequency (**TFIDF**)

BAG OF WORDS (BOW):

Bag of words is the simplest and yet quite an effective technique to vectorize text data in NLP. It consists of creating an ordered lookup table or dictionary of all the words contained in the corpus (ie: Vocabulary) and then representing each document in the corpus in terms of this dictionary. The representation could be in terms of word occurrence (in which case it is called Binary Bag of words) or in terms of word count.

For example, let's say that the corpus has a vocabulary of 10,000 unique words. Firstly the vocabulary is pre-ordered in a specific sequence (generally alphabetically) and then each document is represented with reference to this specifically ordered vocabulary as a 10,000 dimensional vector, by filling the corresponding word location on this vector with its word count (basic BOW) or with the numbers 0 or 1 representing the absence or presence of the word as shown in the image below:



TERM FREQUENCY INVERSE DOCUMENT FREQUENCY:

The matrix structure of **TF IDF** is the same as that shown above for BOW except for the values that the cells contain. Here, the cells contain a value that is the product of the two values described below:

1. **Term Frequency (tf):** The term frequency of a word with respect to a particular document, is simply the number of times a particular word occurs within that document.
2. **Inverse document frequency (idf):** The inverse document frequency of a word is defined as shown below:

$$idf = \log\left(\frac{n}{df}\right)$$

Where **df** is the document frequency of that word (ie: the count of the number of documents in the corpus, in which that word is present) and **n** is the total number of documents in the corpus. The inverse document frequency diminishes the weight of terms that occur very frequently in the corpus and increases the weight of terms that occur rarely. The log function is basically used here to scale the value of the ratio (n)/(df) and make it comparable to the value of the term frequency.

Since TF IDF incorporates global information (**idf**) as well as local information (**tf**), it is considered as a better alternative to BOWs in most cases.

UNIGRAM, BIGRAM AND N-GRAMS:

In most languages, each word in a sentence is by nature codependent on the words that surround it and the meaning/information expressed by the sentence is encoded within the correlations of the words it is composed of. As can be understood from the previous discussions on bag of words and TF IDF, we know that these techniques do not take into

account the semantic context of the words in a sentence. N-grams is a way to incorporate this aspect into these text vectorization techniques.

N-Grams are phrases cut out of a sentence with n consecutive words. So instead of viewing a sentence just as composed of individual words, we view it as composed of a set of equi-length phrases and then apply the same rules while computing the BOW or TF IDF values. Consider the following sentence: “Machine Learning is a fascinating subject”. Let us say this sentence belongs to a corpus we want to vectorize. If we want to compute the BOW or TF IDF using N-grams, where n is equal to 2 (ie: Bigram), then this sentence and all other sentences in the corpus will be modified into the a set of word tuples, such as: “(Machine Learning) (Learning is) (is a) (a fascinating) (fascinating subject) (subject Machine)”. Each of these tuples will then be considered as a single word and BOW/TF IDF is then computed as before.

N-grams is an effective way to encode semantic context, but it comes with a caveat that it increases the dimensionality of the vector representations and thus the computational requirements to process them.

THE AMAZON FOOD REVIEWS DATASET:

For the sake of demonstration we will use the Amazon Food Reviews dataset, which basically consists of customer reviews of the food products sold at Amazon in textual form. We will implement a binary classification system using the text vectorization techniques discussed above, which will serve as an introduction into the world of NLP.

The amazon dataset consists of around 5.7 lakh entries consisting of ten features (columns). We load the data into a pandas dataframe as shown below:

```
# LOADING SQLITE DATA INTO PANDAS DATAFRAME
con = sqlite3.connect(data_path + 'amazon_fine_food_reviews_database.sqlite')

%time df = pd.read_sql_query("""SELECT * FROM Reviews""", con)
df.info()
```

```
 #   Column                  Non-Null Count   Dtype
---  ------                  --------------   -----
 0   Id                      568454 non-null  int64
 1   ProductId               568454 non-null  object
 2   UserId                  568454 non-null  object
 3   ProfileName             568454 non-null  object
 4   HelpfulnessNumerator    568454 non-null  int64
 5   HelpfulnessDenominator  568454 non-null  int64
 6   Score                   568454 non-null  int64
 7   Time                    568454 non-null  int64
 8   Summary                 568454 non-null  object
 9   Text                    568454 non-null  object
```

Out of all the features present in the dataset, we are interested only in four features for the sake of our purposes. They are:

1. **ProductId:** is basically amazon's code for the products sold online
2. **Time:** Refers to the time at which the customer gave the product review
3. **Text:** Refers to the product review
4. **Score:** refers to the rating (1 to 5 stars)

We create a new dataframe with only the relevant data and rename the columns to our convenience as shown below:

```
time = pd.to_datetime([int(i) for i in df.Time.values], unit = 's')
prod = df.ProductId
reviews = df.Text
ratings = df.Score

df_nlp = pd.DataFrame().assign(time = time, prod_id = prod,
                               reviews = reviews, ratings = ratings)

df_nlp.head()
```

	time	prod_id	reviews	ratings
0	2011-04-27	B001E4KFG0	I have bought several of the Vitality canned d...	5
1	2012-09-07	B00813GRG4	Product arrived labeled as Jumbo Salted Peanut...	1
2	2008-08-18	B000LQOCH0	This is a confection that has been around a fe...	4
3	2011-06-13	B000UA0QIQ	If you are looking for the secret ingredient i...	2
4	2012-10-21	B006K2ZZ7K	Great taffy at a great price. There was a wid...	5

DATA PREPROCESSING:

Real world raw data is generally quite messy. It is found to be consisting of missing values, wrong entries and duplicated data. This is more so for textual data. Raw text data has to be first transformed into a more usable form before any kind of vectorization or machine learning can be performed on it. The code shown below transforms the raw data described earlier into a more usable form.

```
1  df_nlp.reviews = df_nlp.reviews.str.lower()
2  df_nlp.prod_id = df_nlp.prod_id.str.lower()
3
4  df_nlp.fillna(' ', inplace = True)
5  df_nlp.drop_duplicates(['reviews'], inplace = True)
6
7  regex_pat_1 = re.compile(r'[^a-zA\n]') #---Matches all chars that are not alpha numeric
8  regex_pat_2 = re.compile(r'\s+') #---------Matches all excess spaces eg: '  '
9
10
11 %time df_nlp.reviews = df_nlp.reviews.str.replace(regex_pat_1, ' ').str.replace(regex_pat_2, ' ')
12
13 df_nlp.index = range(len(df_nlp))
14 df_nlp.sample(5)
```

A B C D

```
CPU times: user 14 s, sys: 201 ms, total: 14.2 s
Wall time: 14.2 s
```

Code block **A** performs the following tasks:

1. Converts all text to non capitalised format.

Code block **B** performs the following tasks:

1. Replace missing values with an empty string/space.
2. Drops all duplicate entries with respect to reviews.

Code blocks **C & D** together perform the following tasks:

1. Block C uses the regex library to define patterns (regex_pat_1 & regex_pat_2), which need to be removed from the specified textual data:
   - Regex_pat_1 is a pattern that matches any non alphanumeric string data.
   - Regex_pat_2 is a pattern that matches any excess white space in string data.
2. Block D removes any textual data that match the patterns described in point 1.

The result of applying the code shown above, is that we transform the raw data to the format shown below:

	time	prod_id	reviews	ratings
98730	2010-11-08	b0030f70lo	healthy treats for our standard american eskim...	5
255377	2012-07-27	b005bgrbm8	for those saying it doesn t pass the taste tes...	5
4095	2012-10-11	b001ew5yqs	this icicle performs flawlessly with very good...	5
124853	2012-07-09	b005grcwdu	lavazza coffee is the best coffee ever we love...	5
188737	2012-08-31	b0032jwe4q	i like my tea strong i normally drink pg tips ...	3

REMOVING LESS IMPORTANT PRODUCTS:

We then filter the data set and use only those products that were bought/reviewed at least twenty times. This is done using the function **fn_df_nlp_top_prods** as shown below:

```
def fn_df_nlp_top_prods(df_nlp, freq_thresh = 20):

    condition = df_nlp.prod_id.value_counts() >= freq_thresh
    df_freqs = df_nlp.prod_id.value_counts()[condition]
    list0_top_prod_ids = list(df_freqs.index )

    list0_filtered_idxs = []
    pbar = ProgressBar()

    for prod_id in pbar(list0_top_prod_ids):

        filtered_idxs = df_nlp[df_nlp.prod_id == prod_id].index
        list0_filtered_idxs += list(filtered_idxs)

    list0_filtered_idxs = list(set(list0_filtered_idxs))

    return df_nlp.iloc[list0_filtered_idxs]
```

```
df_relevant_prods = fn_df_nlp_top_prods(df_nlp)

df_relevant_prods.sample(10)
```

```
100% (3467 of 3467) |####################| Elapsed Time: 0:01:32 Time:  0:01:32
```

	time	prod_id	reviews	ratings
228233	2011-03-26	b001eo772a	a great coffee in pre ground i don t have a co...	5
221818	2012-02-24	b0051coph6	span class tiny length mins br br span here a...	4
209569	2011-02-20	b004bklhos	amazon offered apple cinnamon and golden honey...	5
305287	2011-09-22	b002gq3idc	i grew up within an indian community and have ...	5
21336	2009-12-05	b000kv61fc	i wanted to use this as an alternative to a ko...	2
143608	2011-06-15	b001b3jal0	i really really love pop chips they are much l...	4
37456	2011-03-07	b00017ley8	i cannot believe the vendor had the nerve to s...	1
175234	2008-01-07	b0009xuax0	great products easy to deal with i order from ...	5
355251	2011-02-24	b000n33nsu	i first purchased erin baker s breakfast cooki...	5
245210	2012-06-25	b001j8h1e0	my dogs love these and its great to finally fi...	5

```
df_relevant_prods.shape
```

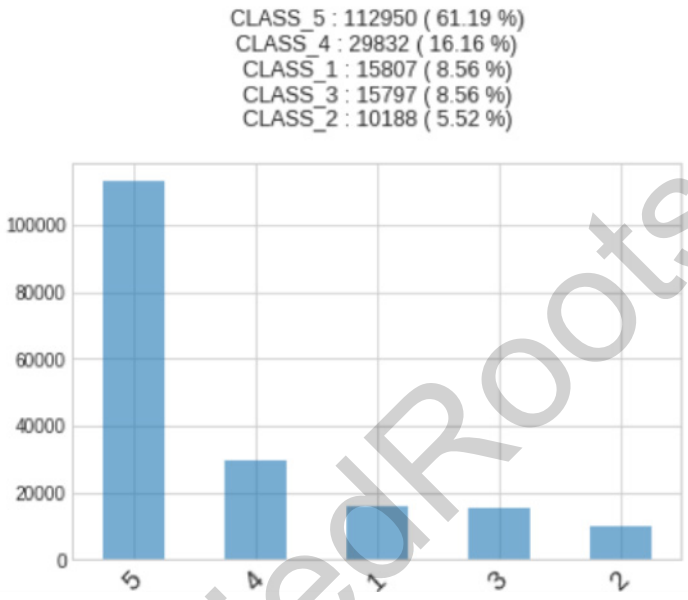
```
(184574, 4)
```

As can be seen from the shape of the data above, the data set now contains only around 1.8 lakh entries. We shall use this data set for our purposes.

CHECKING THE DISTRIBUTION OF PRODUCT RATINGS:

We check the distribution of the ratings as shown below:

```
y = df_relevant_prods.ratings.values

ai.fn_distr_labels_clf(y)
```

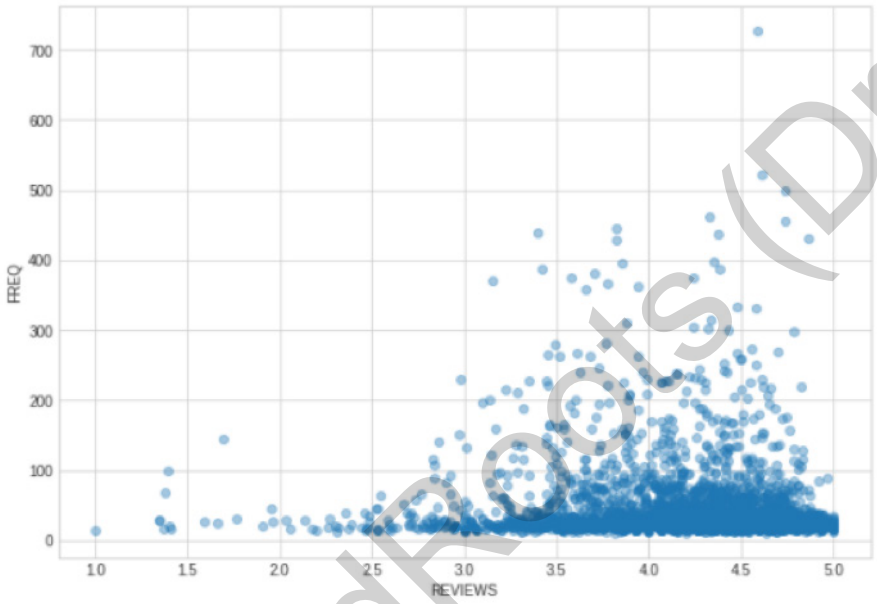


VISUALIZING INTERACTION BETWEEN REVIEWS AND FREQ OF REVIEWS:

```
df_prod_ratings = df_relevant_prods.groupby('prod_id').mean()
df_prod_ratings = df_prod_ratings.sort_values(by = 'ratings', ascending = False)
df_prod_freq = df_relevant_prods.prod_id.value_counts()
df_prod_rating_freq = pd.concat([df_prod_ratings, df_prod_freq], axis = 1)
df_prod_rating_freq.columns = ['ratings', 'freq']

df_prod_rating_freq.head()
```

	ratings	freq
b000leq0qs	5.0	21
b0016bs3bk	5.0	25
b001iz9me6	5.0	30
b000f3yepo	5.0	20
b0029o0xgq	5.0	30

```
plt.figure(figsize = (10, 7))
plt.scatter(df_prod_rating_freq.ratings, df_prod_rating_freq.freq, alpha = 0.4)
plt.xlabel('REVIEWS')
plt.ylabel('FREQ')
plt.show()
```



From the scatter plot shown above, we see that the top products are those with a review greater than or equal to 3.5 and number of reviews greater than or equal to 400. We isolate these top products as shown below:

IDENTIFYING WHAT THE TOP PRODUCTS ARE:

```
cond_1 = df_prod_rating_freq.ratings >= 3.5
cond_2 = df_prod_rating_freq.freq >= 400

df_top_prods = df_prod_rating_freq[cond_1 & cond_2]
df_top_prods
```

	ratings	freq
b000nmjwzo	4.867749	431
b003b3oopa	4.737475	499
b001eo5q64	4.736264	455
b002qwp89s	4.609195	522
b007jfmh8m	4.590096	727
b000ubd88a	4.376147	436
b0013nugde	4.331169	462
b0090x8ipm	3.825175	429
b005k4q37a	3.820225	445

We then identify what the actual top products are, by randomly selecting text that correspond to the product_ids of the top products shown and inspecting them above as shown below:

```
cond = df_relevant_prods.prod_id == df_top_prods.index[0]

df_relevant_prods[cond].reviews.sample(5).values
```

```
array(['when we first started using this stuff we were on the subscribe and save program getting pound bags ever
       'i love this mix because i ve been able to use it to recreate some old family recipes the ones grandma wo
       'i have been using this mix for years for no other reason than it is superb when you make pancakes or waf
       'i love this mix not only are the pancakes and muffins great but almost every recipe tried has turned out
       'excellent for pancakes and baking i have tried almost all the recipes on the bag and they all taste grea
```

From the sample reviews above, we infer that the **top most product** with an average rating of 4.86 and frequency of 431 reviews is pancake or some kind of **cake baking mix**.

```
cond = df_relevant_prods.prod_id == df_top_prods.index[1]

df_relevant_prods[cond].reviews.sample(5).values
```

```
array(['this is a must have if you re a college student cooking for yourself i can use this on the pan i can use tl
       'i love this stuff i use it on my skin and it drinks it up i put it in my hair before washing and my hair l
       'i was rather disappointed with this product first it arrived totally melted and liquified to the company s
       'i decided to order this product after reading a few books about the benefits of coconut oil i received thi
       'i use this organic coconut oil in my sons hair its fantastic smells great doesn t leave his hair dry no su
```

From the sample reviews above, we infer that the **second top most** product with an average rating of 4.74 and frequency of 499 reviews is some kind of **coconut oil.**

```
cond = df_relevant_prods.prod_id == df_top_prods.index[2]

df_relevant_prods[cond].reviews.sample(5).values
```

```
array(['i bought this product for my hair because of other people s reviews my hair was brittle and looked horribl
       'great product quick shipment great price i love love love coconut oil and it s quite pricey i should have
       'i got this because i had read on another website that this product was great for using in place of butter
       'i had heard about the benefits of coconut oil from a friend who swears by coconut water which is quite pos
       'i knew my coconut oil had arrived because of the dark oily spot spreading over the outside of the box the
```

From the sample reviews above, we infer that the **third top most** product with an average rating of 4.73 and frequency of 455 reviews is again some kind of **coconut oil.**

```
cond = df_relevant_prods.prod_id == df_top_prods.index[3]

df_relevant_prods[cond].reviews.sample(5).values
```

```
array(['i have never seen my dogs as happy as when they seem me take out the greenies bag they are pri
       'these do help to keep my dogs teeth clean and they are just the right size for my lb chihuahua
       'my little dog loves greenies chews we were first introduced to this dental treat at our vet s
       'einstein our puppy loves the greenies he get s one at night before bed as a bribe for going in
       'when we opened the box we found it to be full of small worms my husband was able to remove the
```

From the sample reviews above, we infer that the **fourth top most** product with an average rating of 4.61 and frequency of 522 reviews is some kind of **dog food.**

```
cond = df_relevant_prods.prod_id == df_top_prods.index[4]

df_relevant_prods[cond].reviews.sample(5).values
```

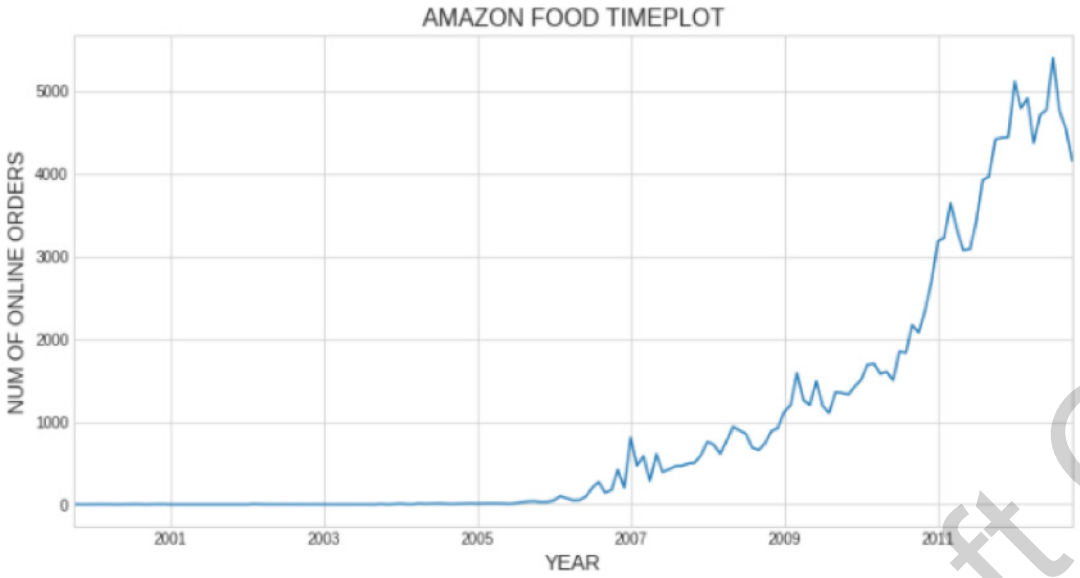
```
array(['i loved these cookies they are chewy and moist and the raisins are not hard or gummy they are very fresh a
       'i received a quaker soft baked oatmeal raisin cookie from influenster to try and review i shared this with
       'my daughter loved my free sample of this cookie from influenster she goobled it right down which i didn t
       'delicious and soft treat perfect to store in a purse or diaper bag my kids loved them and i enjoyed them a
       'they are great tasty and delicisous we even tried the banana nut ones they were fantastic we loved them my
```

From the sample reviews above, we infer that the **fifth top most** product with an average rating of 4.59 and frequency of 727 reviews is some kind of **cookie food.**

NUMBER OF ONLINE ORDERS/REVIEWS OVER THE YEARS:

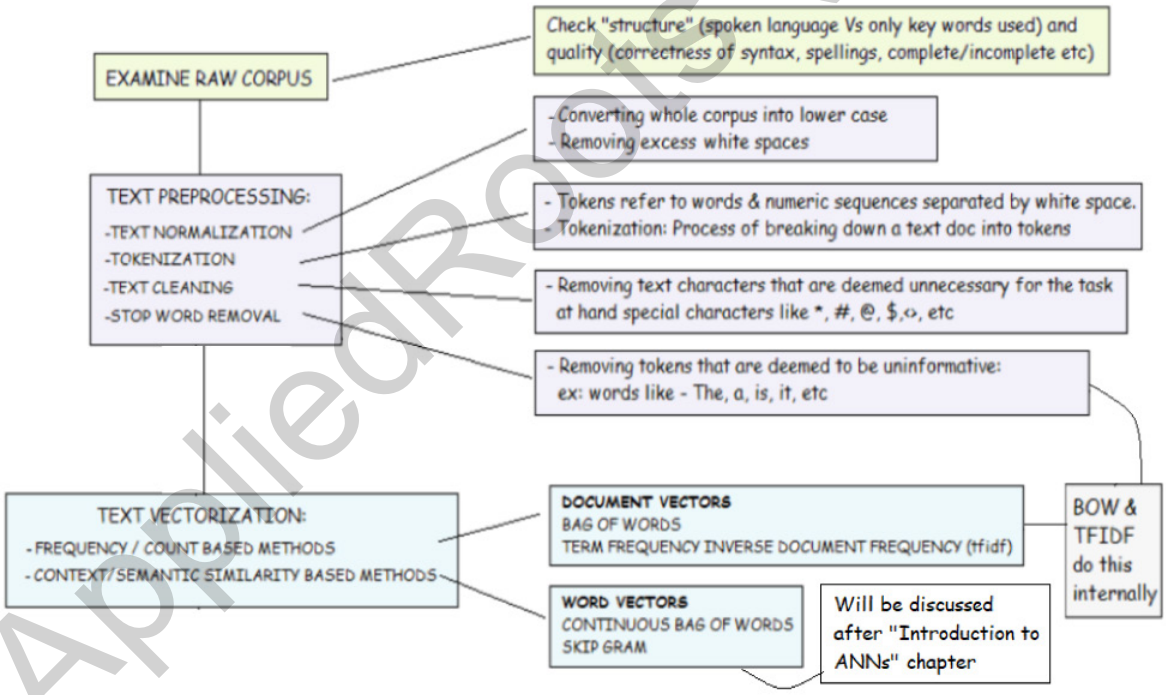
We can check how the number of online reviews/orders have changed over the years, as shown below:

```
freq = [1 for i in range(len(df_tr))]
df_timestamp = pd.DataFrame().assign(prod_id = df_tr.prod_id, freq = freq)
df_timestamp.index = df_tr.time

fig, ax = plt.subplots(figsize=(12,6))

df_timestamp.freq.resample('M').sum().plot(ax=ax)
ax.set_title('AMAZON FOOD TIMEPLOT', fontsize = 16)
ax.set_ylabel('NUM OF ONLINE ORDERS', fontsize = 14)
ax.set_xlabel('YEAR', fontsize = 14)
ax.xaxis.set_ticks_position('bottom')
plt.show()
```



OVERVIEW OF NLP PIPELINE:

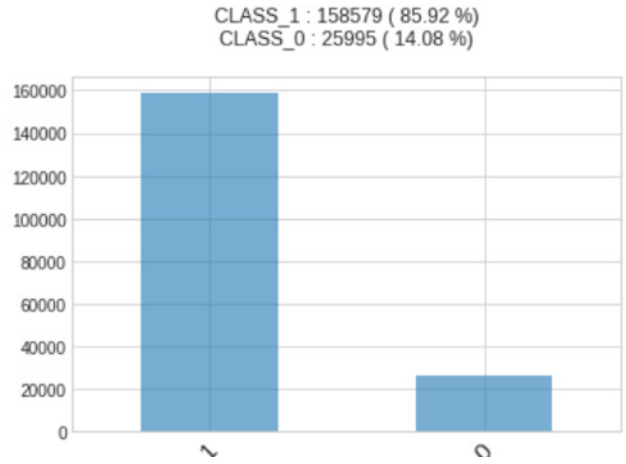
The image below provides an overview of a basic natural language processing pipeline. The Bag of words and Term Frequency Inverse Document Frequency techniques generally benefit from stop word removal since they are "count' based techniques.



CONVERTING LABELS FROM RATINGS TO POSITIVE & NEGATIVE CLASS:

```
pos_neg_ratings = [1 if int(i) >=3 else 0 for i in df_relevant_prods.ratings]

df_Xy = df_relevant_prods.drop('ratings', axis = 1).assign(ratings = pos_neg_ratings)
df_Xy.head()
```

```
y = df_Xy.ratings.values

ai.fn_distr_labels_clf(y)
```



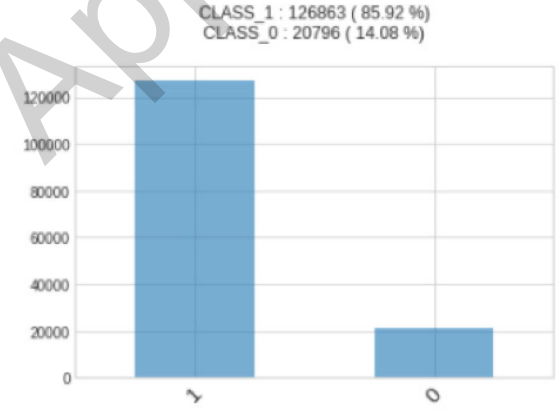
TRAIN – TEST SPLIT:

```
idxs_tr, idxs_ts = ai.fn_tr_ts_split_clf(df_Xy, ts_size = 0.2)

df_nlp_tr = df_Xy.iloc[idxs_tr]
df_nlp_ts = df_Xy.iloc[idxs_ts]

df_nlp_tr.index = range(len(df_nlp_tr))
df_nlp_ts.index = range(len(df_nlp_ts))

df_nlp_tr.shape, df_nlp_ts.shape
```

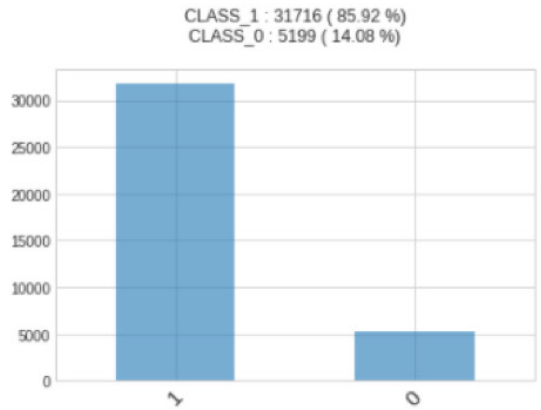
```
((147659, 3), (36915, 3))
```

CHECKING CLASS DISTRIBUTION ACROSS TRAIN AND TEST SETS:

```
y = df_nlp_tr.ratings.values

ai.fn_distr_labels_clf(y)
```




```
y = df_nlp_ts.ratings.values

ai.fn_distr_labels_clf(y)
```



TEXT VECTORIZATION USING BOW AND TFIDF:

As discussed previously, Bag of words and Term frequency Inverse Document Frequency techniques are count based techniques. These can be implemented using the scikit learn library. The image below describes the BOW class (ie: CountVectorizer) and the default values of some its most relevant parameters:

```
CountVectorizer(stop_words = None,
                n_gram_range = (1, 1),
                min_df = 1,
                max_df = 1.0,
                max_features = None)
```

BOW PARAMETERS:

Stop_words: This parameter defines if stop word removal needs to be carried out. If it is set to None, it is avoided, if it is set to '**english**', common non informative words like 'The', 'is', but', 'again', 'there', 'about', etc are removed. The scikit learn library has its own set of around 300 stop words, which it removes from the text it is processing, if this parameter is set to 'english'.

n_gram_range: This parameter defines the kind of n_grams to be computed. It is specified in the form of tuple indicating the minimum and maximum N-grams to be computed. For instance, specifying the tuple (1, 3) indicates that BOWs need to be computed using single words, two word phrases and three word phrases.

min_df: This parameter defines the lower threshold of a word's document frequency for it to be considered for the BOW computation. It is defined as an integer value. Thus a min_df value of 5 would indicate that, if a word has to be considered, it has to at least exist in 5 documents.

max_df: This parameter defines which words to include in the BOW computation based on its document frequency. It has to be a float value greater than zero and upto 1.0 indicating the percentage of the maximum number of documents the word being considered can exist in, for it to be considered for the BOW computation. Thus a max_df value of 0.9 would indicate that words that exist in more than 90% of the documents should not be considered.

max_features: If not None, it tells the model to build a vocabulary that only considers the top max_features (ordered by top term frequency value of words across the corpus).

The class for computing **TF IDF** vectors (ie: **TfidfVectorizer**) also contains the same relevant parameters defined above.

BOW CODE:

Shown below is the code to create BOW vectors using scikit learn's **CountVectorizer** class:

```
corpus = df_nlp_tr.reviews.values

kwargs = dict(min_df = 10, max_features = 3000,
              ngram_range = (1, 1), stop_words = 'english')
BOW_vectorizer = CountVectorizer(**kwargs).fit(corpus)
tr_BOW_matrix = BOW_vectorizer.transform(corpus)
```

The execution of the above code gives us a scipy.sparse.csr_matrix (**bow_matrix**). The scipy sparse matrix format is a compressed format for sparse matrices, which stores the values in terms of non-zero values and their corresponding matrix index location (**non_zero_value : location**) pairs, instead of the entire matrix structure itself. Since most of the values are zero, this format achieves a high degree of data compression. Note that we have performed stop word removal on the text prior to the vectorization, by setting the **stop_words** parameter in the CountVectorizer instance above to '**english**'.

Note that we set the **max_features** class parameter to 3000. This makes the model choose only the top 3000 words (in terms of their term frequency scores), to be a part of its vocabulary. Hence the shape of **bow_matrix** is 147659 x 3000 indicating that there are 3000 words in the vocabulary and thus there are that many features for each of the 147659 vectorized documents.

TF IDF CODE:

The process for performing TF IDF text vectorization is similar to that used for BOW. Shown below is the code to create TF IDF vectors using scikit learn's **TfidfVectorizer** class:

```
corpus = df_nlp_tr.reviews.values

kwargs = dict(min_df = 10, max_features = 3000,
              ngram_range = (1, 1), stop_words = 'english')
TFIDF_vectorizer = TfidfVectorizer(**kwargs).fit(corpus)
tr_tfidf_matrix = TFIDF_vectorizer.transform(corpus)

tr_tfidf_matrix.shape
```

```
(147659, 3000)
```

We then use the **TFIDF_vectorizer** derived from the trainset to create TF IDF vectors for the testset as shown below.

```
ts_tfidf_matrix = TFIDF_vectorizer.transform(df_nlp_ts.reviews.values)

ts_tfidf_matrix.shape
```

```
(36915, 3000)
```

DIMENSIONALITY REDUCTION AND DATA VISUALIZATION:

We will use the TFIDF vector representation of the corpus (ie: customer reviews), for further purposes of classification. We dimensionally reduce its 3000 dimensional vectors to 100 dimensional vectors using the TruncatedSVD class (ie: Find first 100 principal components using PCA principle) which is done as shown below:

```
from sklearn.decomposition import TruncatedSVD
```

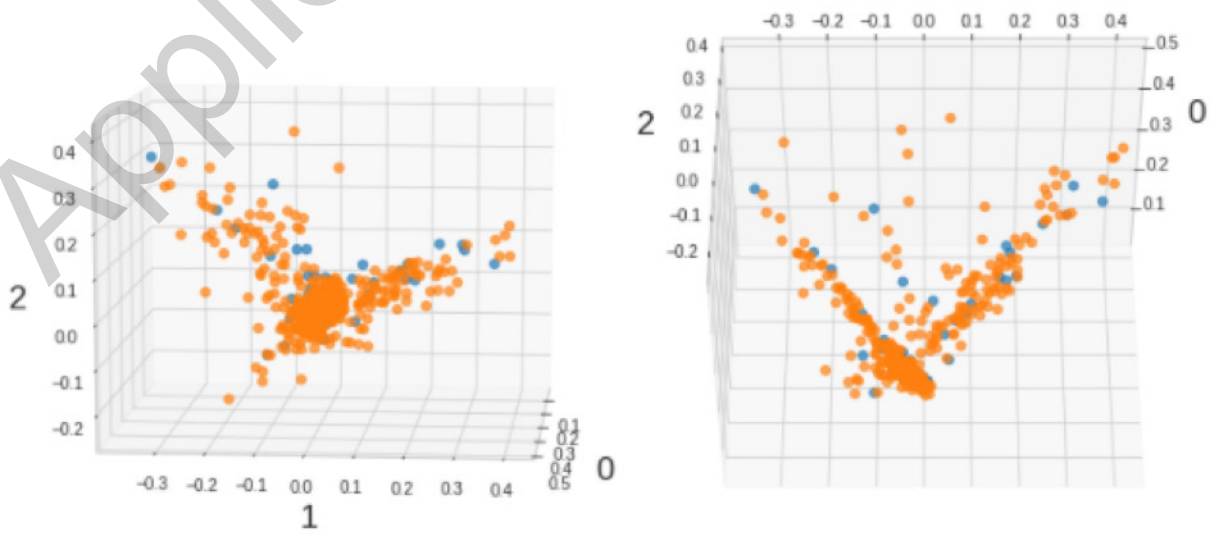
```
svd = TruncatedSVD(n_components=100, n_iter=7, random_state=42)
svd.fit(tr_tfidf_matrix)
X_train_ = svd.transform(tr_tfidf_matrix)
X_test_ = svd.transform(ts_tfidf_matrix)
```

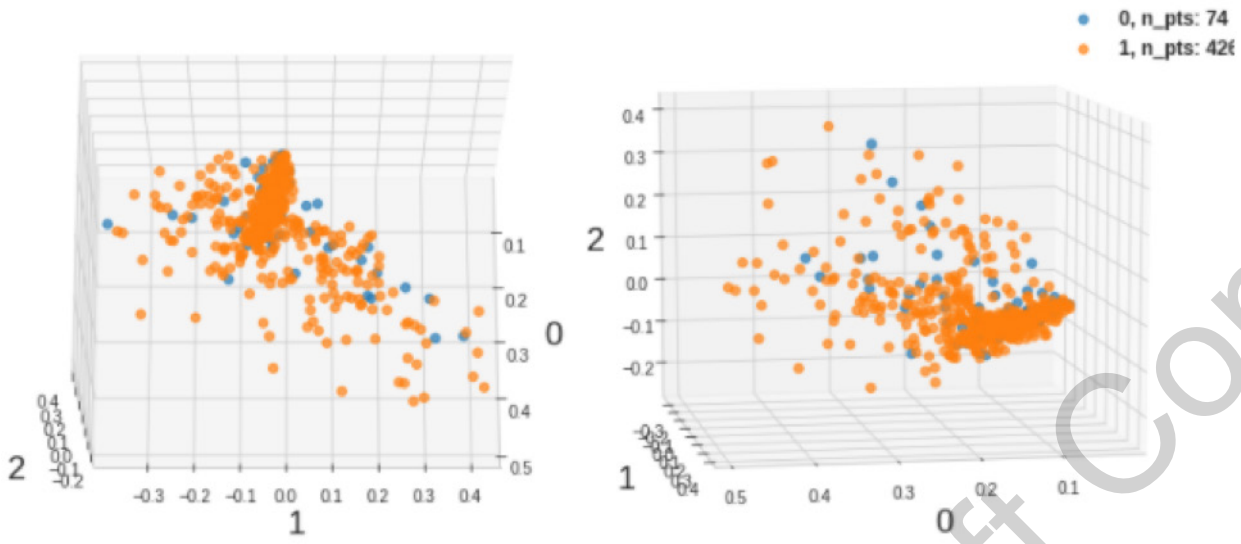
```
df_tr = pd.DataFrame(X_train_).assign(labels = df_nlp_tr.ratings)
df_ts = pd.DataFrame(X_test_).assign(labels = df_nlp_ts.ratings)

df_tr.shape, df_ts.shape
```

```
((147659, 101), (36915, 101))
```

VISUALIZING THE DIMENSIONALLY REDUCED DATA:

```
ai.fn_plot_3d_clf(df_tr.sample(500), alpha = 0.7)
```





K FOLD CROSS VALIDATION AND HYPERPARAMETER TUNING:

We perform grid search with K fold cross validation using the **Logistic Regression** classifier as shown in the code below. We use the '**saga**' solver since it works for all regularization (ie: L1, L2 and elastic net).

```
from sklearn.linear_model import LogisticRegression

model_class = LogisticRegression
df_tr_ = df_tr
parameter_grid = dict(penalty = ['l2', 'l1', 'elasticnet'],
                      C = [10**i for i in range(-15, 15)],
                      solver = ['saga'],
                      class_weight = ['balanced'],
                      max_iter = [10_000])

z = ai.fn_kfoldcv_clf_binary(df_tr_, model_class,  parameter_grid)

df_kfoldcv_gridsearch, dict0_model_instances, listo_invalid_models = z
```

```
100% (90 of 90) |########################| Elapsed Time: 0:10:36 Time:  0:10:36
```

```
regd_columns = 'tr_prec_1 tr_rec_1 tr_prec_0 tr_rec_0 diff_rec_1 diff_rec_0'.split()
df_kfoldcv_gridsearch.loc[:, regd_columns].describe()
```

	tr_prec_1	tr_rec_1	tr_prec_0	tr_rec_0	diff_rec_1	diff_rec_0
count	60.000000	6.000000e+01	60.000000	60.000000	6.000000e+01	60.000000
mean	0.954787	6.531536e-01	0.385552	0.747867	3.751756e-04	0.001728
std	0.020761	2.219096e-01	0.127700	0.157615	3.199168e-04	0.001216
min	0.906108	1.182378e-11	0.140838	0.333333	-2.483095e-04	-0.000192
25%	0.953054	6.666667e-01	0.360953	0.755530	-3.941181e-12	0.000000
50%	0.959795	7.666774e-01	0.361006	0.804097	5.241717e-04	0.002621
75%	0.959802	7.666863e-01	0.361028	0.804121	5.369807e-04	0.002651
max	1.000000	7.747964e-01	0.713613	1.000000	1.561921e-03	0.003390

We then filter the best models as shown below:

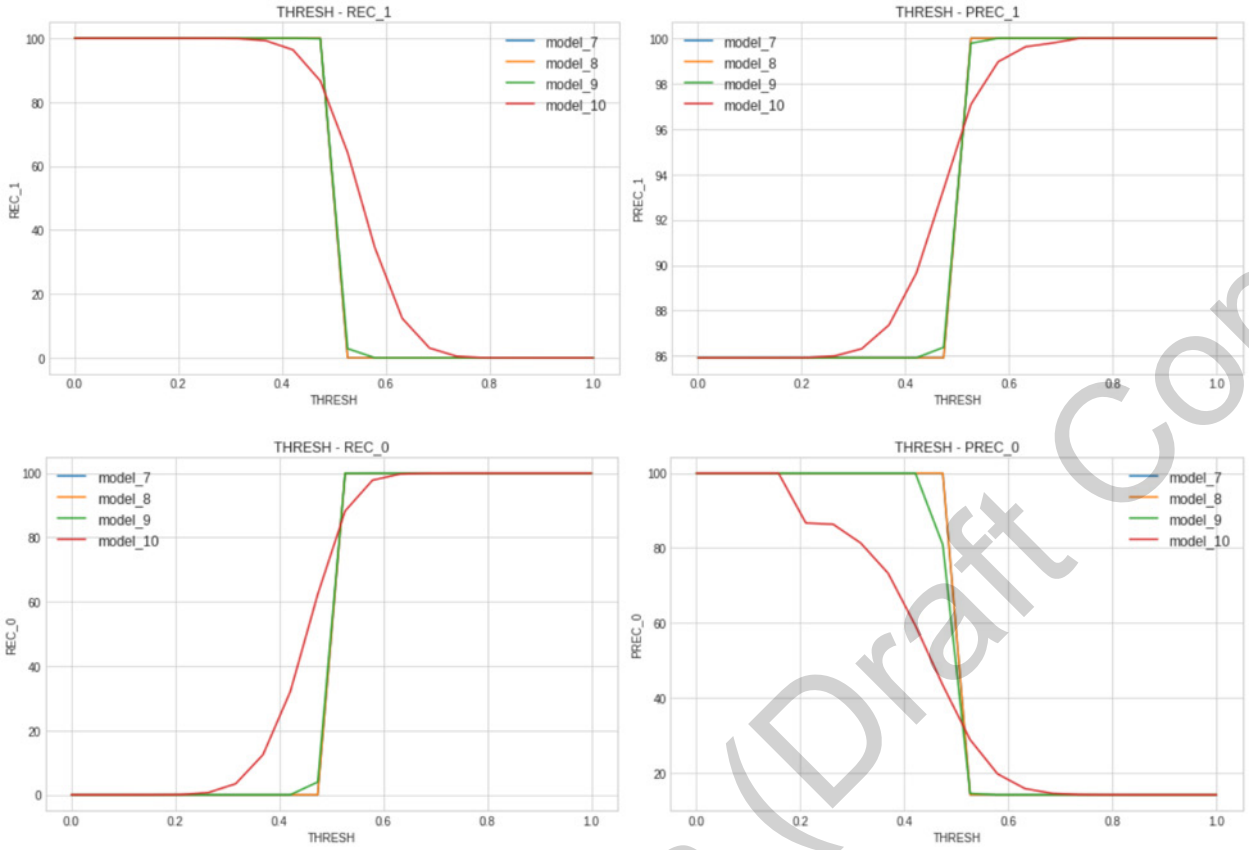
```
dff = df_kfoldcv_gridsearch
df1 = dff[dff.diff_rec_0 < 0.06]
df2 = df1[df1.tr_rec_0 > 0.7]
df_filtered = df2[df2.tr_rec_1 > 0.77]
df_filtered.loc[:, regd_columns]
```

	tr_prec_1	tr_rec_1	tr_prec_0	tr_rec_0	diff_rec_1	diff_rec_0
model_7	0.950780	0.774796	0.354754	0.755314	0.000733	0.001082
model_8	0.950823	0.774611	0.354651	0.755602	0.000800	0.001226
model_9	0.950972	0.774398	0.354690	0.756444	0.000855	0.001154
model_10	0.952470	0.772266	0.355085	0.764907	0.000276	0.001539

CHECK MODEL PERFORMANCE USING PRECISION & RECALL CURVES:

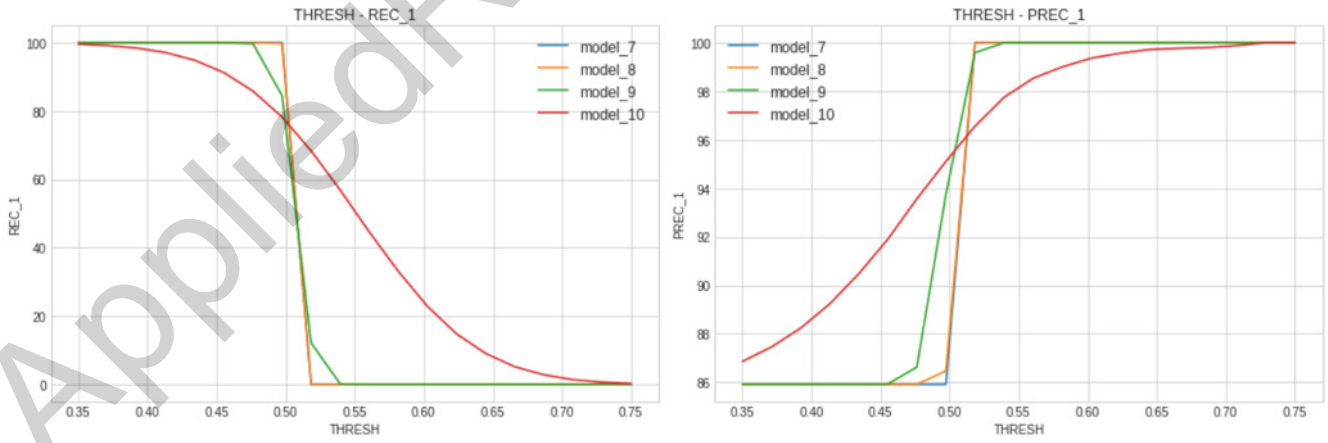
We inspect the precision and recall curves for the top models as shown below.

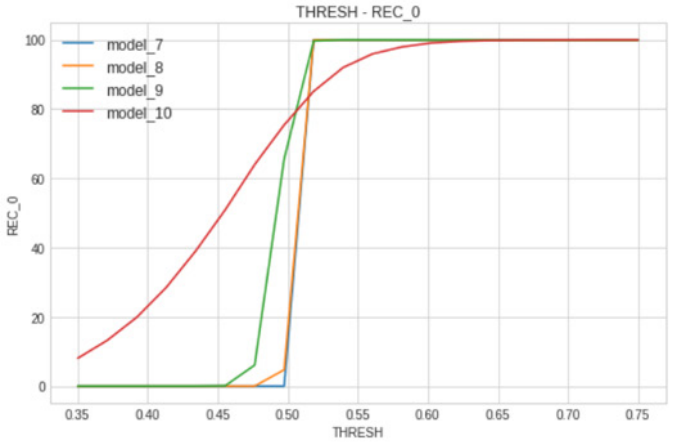
```
df_tr_standard, df_ts_standard, scaler = ai.fn_standardize_df(df_tr, df_ts)

list0_model_names = list(df_filtered_cv.index)
X_train, y_train = df_tr_standard.iloc[:, :-1].values, df_tr_standard.iloc[:, -1].values
list0_models = [dict0_model_instances[i].fit(X_train, y_train) for i in list0_model_names]

list0_thresholds = np.linspace(0, 1, 20)
legend = list0_model_names

ai.fn_performance_models_data(list0_models, df_tr_standard, list0_thresholds, legend)
```

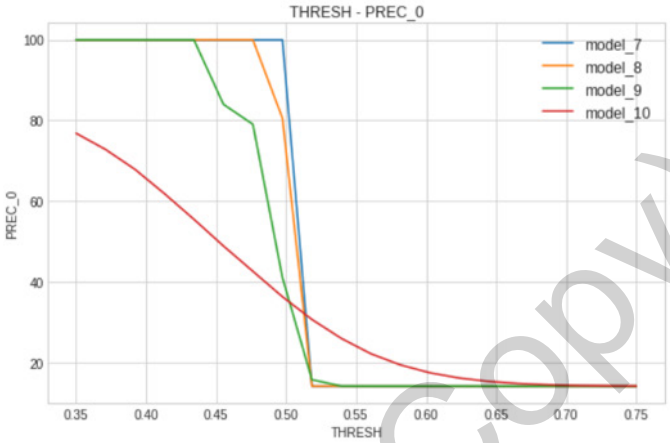


We choose to have a closer look at the threshold range between 0.35 and 0.75 as shown below

```
1  listO_thresholds = np.linspace(0.35, 0.75, 20)
2  legend = listO_model_names
3
4  %time ai.fn_performance_models_data(listO_models, df_tr_standard, listO_thresholds, legend)
```







Assuming that we are optimizing for the recall of the negative reviews (class 0), we inspect the plots shown above and choose model_10 and threshold value of 0.51.

We then check how well the chosen model generalizes as shown below:

```
df_Xy_ = df_tr_standard
model_ = dictO_model_instances['model_10'].fit(X_train, y_train)
thresh = 0.51

ai.fn_test_model_binary_clf(df_Xy_, model_, threshold_class_1 = thresh)
```

```
--------------------
LOGLOSS : 0.6048
ACCURACY: 73.886
--------------------
```

	prec	rec
class_0	32.811	81.520
class_1	95.996	72.635

```
df_Xy_ = df_ts_standard

ai.fn_test_model_binary_clf(df_Xy_, model_, threshold_class_1 = thresh)
```

```
--------------------
LOGLOSS : 0.6049
ACCURACY: 74.011
--------------------
```

	prec	rec
class_0	32.916	81.439
class_1	95.988	72.793

As can be seen from the train and test performances of the model, it generalizes quite well. The model gives us 81% recall on class_0 (negative reviews) and 72% recall of class_1. The precision for the negative class is quite less. It is around 33%, which means 67% of the negative predictions are actually positive. This is because around 28% of the positive reviews have been predicted as negative. We could build a second stage classifier and apply it to the negative predictions of this model, to further improve performance.

5. FUNDAMENTALS OF ARTIFICIAL NEURAL NETWORKS

FUNDAMENTALS OF ARTIFICIAL NEURAL NETWORKS

Artificial Neural Networks (ANNs) are machine learning models whose computational structure was initially arrived at by trying to simulate the working of a neuron, which could be considered as the fundamental unit of the human brain. A neuron basically receives information (signals) from many of its branches and then transforms these multiple inputs into a single output signal and passes it on to another branch. This output branch then forms one of the input branches to another neuron and so on.

The artificial neuron is the fundamental computational unit of an artificial neural network. An artificial neuron has multiple scalar input branches and a single scalar output branch. Each Artificial neuron is associated with:

A. A set of weights, one for each of its input branches (one scalar value per **branch**). The number of input branches is always the same as the dimensionality of the input that it is going to be trained over.
B. A bias value (ie: one scalar value per neuron).

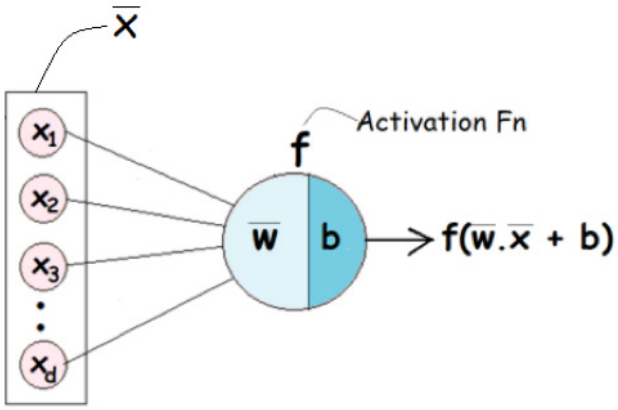
The basic computation associated with an artificial neuron is:

1. The product of the inputs at each of its branches, with the corresponding weights associated with these branches.
2. The sum of all the products got at step 1.
3. The addition of the sum got at step 2 with the bias value.
4. The value obtained from the computations at steps 1,2 & 3, is then passed through a nonlinear function (ex: sigmoid, tanh, ReLU)

Thus, if the input values are viewed as a single 'd' dimensional vector $\bar{\mathbf{x}}$ and the weights associated with each of the input branches as a 'd' dimensional weight vector $\bar{\mathbf{w}}$, then the computation associated with an artificial neuron could be expressed as:

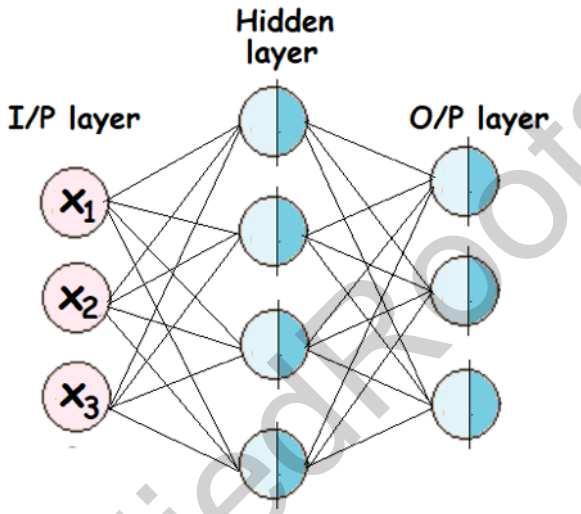
$f(\bar{w}.\bar{x} + b)$ where b is the bias term &
f is the nonlinear function **(ReLU in most modern neural networks)**

The structure and function of an artificial neuron could thus be visually described as shown below. The weights $\bar{\mathbf{w}}$ and bias b can be varied (or updated) to learn a function that produces an approximation of the desired outputs, given the corresponding input vectors. The figure below shows a single input of d dimensions and its association with the neuron.



MULTILAYER PERCEPTRONS (MLPS):

A basic artificial neural network would have the structure/architecture as shown in the image below. Neural networks that have the depicted architecture are called Multi Layer Perceptrons. Here perceptrons is another term for neurons and multilayer refers to the layered architecture of the mode as will be seen further along the chapterl.



The Artificial Neural Network shown above has three layers:

1. An **input layer**, which basically represents the input data (3D in this example).
2. A **hidden layer**, which contains a set of neurons (4 in this example) each acting in parallel, on the same input data $\bar{x}$ and each producing a single output value $\mathbf{f(\bar{w}.\bar{x} + b)}$. Each neuron in the hidden layer is generally associated with its own unique weight vector $\bar{w}$ and bias value **b**.
3. An **output layer** which is reflective of the task that the neural network is performing. In the example shown above, the neural network has 3 neurons in its output layer, this implies that it is designed to perform a 3 class classification problem.

The neurons of the output layer differ slightly from those of the hidden layer. Just like the neurons of the hidden layer, each of the neurons of the output layer generally have unique weight vectors and biases associated with them, but their outputs differ because they do not have any activation function **f**. In other words, they only compute $\bar{w} \cdot \bar{x} + b$.

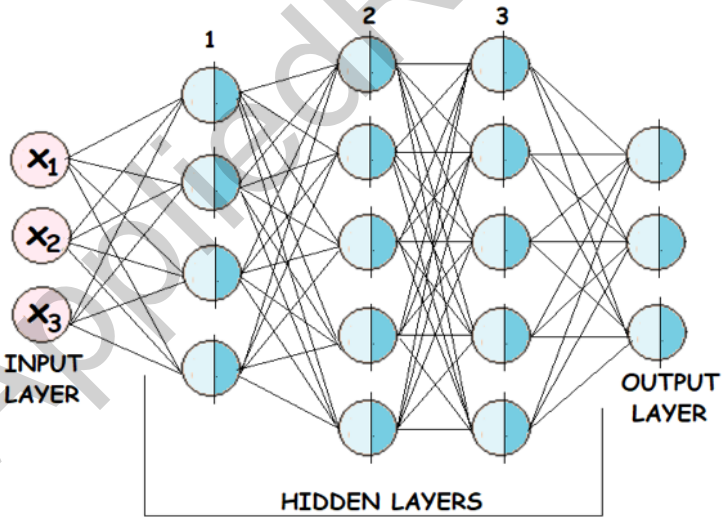
In case of a binary classification task, the output layer will consist of just one neuron and the output of this neuron is passed through a sigmoid function. We then use log loss loss to compute/measure the difference in the predicted output and the desired output (ie: The label).

In the case of a linear regression task, the output layer will consist of just one neuron and the output of this neuron is not passed through any function. We then use squared loss or huber loss to compute the difference in the predicted output and the desired output.

Incase of a multiclass classification task, the output layer will consist of the same number of neurons as the number of classes to be classified. The vector output of this layer would be then passed through a softmax function to convert it into a probability distribution between the classes. We then use cross entropy loss to compute the difference in the predicted output vector and the desired output. The desired output label in this case would be represented in the form of a one hot encoded vector which indicates the probability distribution of the desired output across all classes (ie: 1 for correct class position and zero for all other classes).

LAYER WIDTH AND NETWORK DEPTH:

Artificial neural networks have no theoretical limitations on the width of its hidden layers or on the number of hidden layers that are used between its input layer and output layer. Consider the image shown below:



The neural network shown consists of 3 hidden layers and so it is defined as having a depth of 3. The first hidden layer has a width of 4 and the next two hidden layers have a width of 5. There are no actual scientific methods or rules as how to architect the structure of neural networks, except that the width of the input layer should be same as the dimensionality of the data it is being trained on and that the width of the output layer is defined by the task the network is going to be used for, as explained previously.

Neural networks become very powerful as their depth increases. Modern neural networks which are designed to learn from extremely huge datasets can have depths that exceed more than ten to fifteen layers and hence the term "**Deep Learning**". As the depth of a neural network increases, several factors come into play which hinder the learning process and sometimes even make it impossible to train, such as: gradient explosion problem, vanishing gradient problem and overfitting/data memorization. At the same time, several methods have been designed/discovered to overcome these problems such as: batch normalization, gradient clipping, using specific activation functions that belong to the ReLU family, applying drop-out layers between hidden layers and so on. These advanced topics will be discussed in more detail in another chapter called "Deep Learning". For now, this chapter will serve as an introduction to artificial neural networks. We will use the **Keras** framework, with its default parameters to build basic neural networks and train them to solve the same tasks we have explored in the previous chapters.

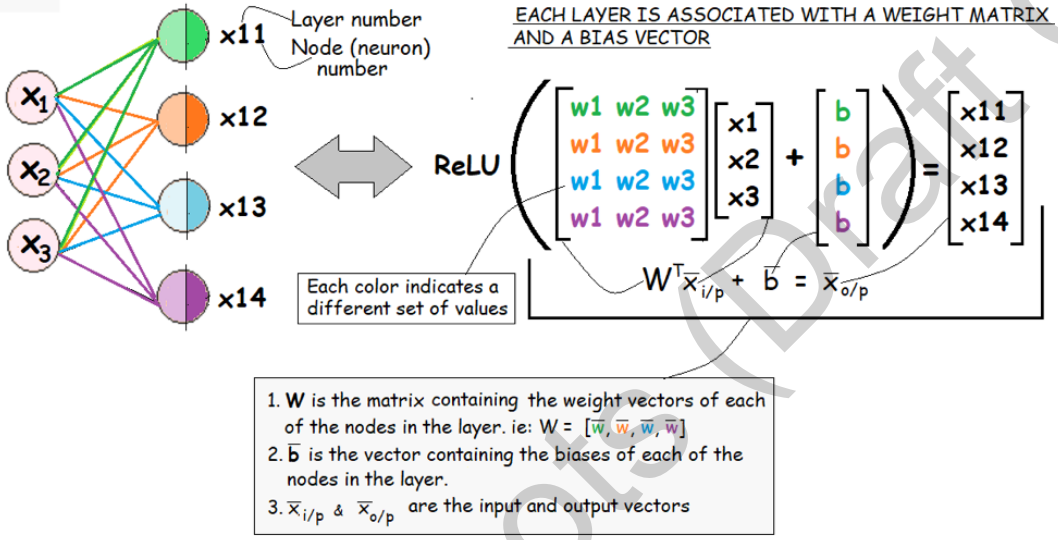
LEARNING THE OPTIMAL SET OF WEIGHTS (FORWARD AND BACKWARD PROPAGATION):

The core aim of training neural networks is to find the optimal set of weights, given a particular neural network configuration (breadth of its layers and number of layers). This Is achieved by using a process similar to the gradient descent algorithm used in Logistic Regression (Note that, Logistic Regression could be viewed as a single layer of neurons without activation functions, having the same breadth as the number of classes). The steps involved in finding the optimal set of weights are outlined below:

1. **WEIGHT INITIALIZATION**: In this step the weights of all the layers of the neural network are initialized using some random initialization scheme that ensures "optimal asymmetry". Asymmetric weight initialization ensures that each neuron learns about a different aspect of the data. This would not be so if the neural network was initialized such that all neurons have the same weights associated with them. The weight initialization scheme should be such that apart from asymmetry, it should also ensure that the weights should neither be too large or too small, and should also have optimal variance. In the early years of Neural networks, the weight initialization strategy was to sample the weights from a Gaussian distribution. Over the years, other strategies were also implemented. This will be touched upon further along in the chapter 'Deep Learning".

Choosing the best weight initialization technique depends on the task at hand. It could be considered as another hyperparameter to be tuned. The details of these initialization techniques will be discussed in more detail in another chapter further along the book as mentioned earlier.

2. **FORWARD PROPAGATION:** A suitably sized chunk of the data is then randomly sampled and fed to the neural network via its input layer. The sampled data is then propagated through all the layers of the neural network and it is finally outputted at the output layer. When interpreted linear algebraically we can express the computation that occurs at each layer as shown below.

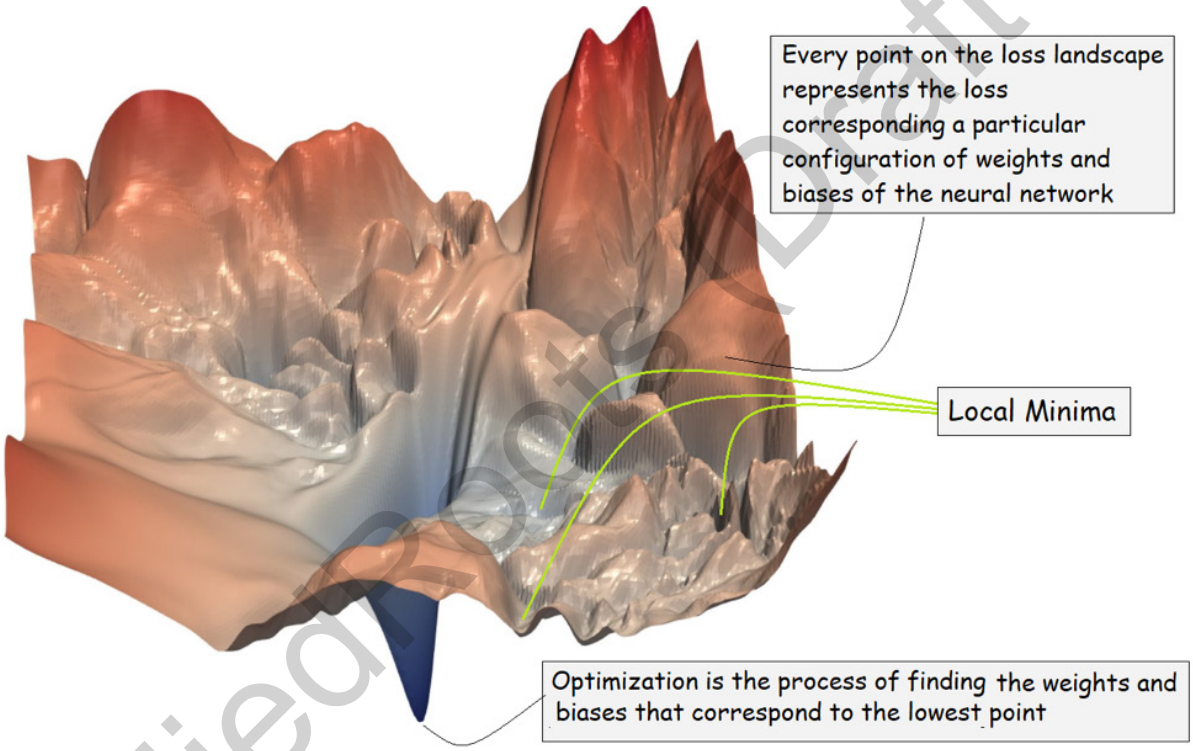


The output of the one layer serves as the input for the next layer and the matrix transformation described above is repeated for all the layers until the last layer. This is forward propagation. A layer is a set of neurons acting parallely on the same input. Each layer is associated with a **weight matrix** (also referred to as **weight kernel**) and a **bias vector**. A layer basically subjects its input vector to a matrix transformation and then passes the resulting transformed vector through a ReLU function. Thus a neural network layer is generally a " vector input - vector output" function. The dimensionality of the output vector is defined solely by the number of neurons used in the layer (Dimensionality of output = the number of neurons in the layer).

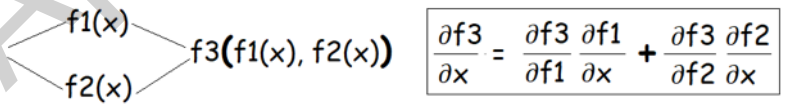
3. **LOSS COMPUTATION:** After a data point $\bar{x}$ is forward propagated through all the layers in the network, we end up with a scalar value or a vector value, dependending on the type of task (Regression, Binary classification or Multiclass classification) the neural network is performing. This represents the predicted output. The difference between the predicted output and the desired output (ie: label) is computed using the appropriate loss function for the task. This process is repeated for all the data points in the randomly sampled data chunk. Since forward propagation of these data points are independent of each other, this operation is trivially parallelizable. The mean loss for the entire chunk of data is then computed.

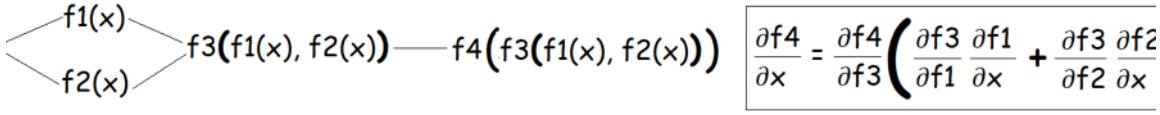
4. **UPDATE WEIGHTS & BIASES (BACKWARD PROPAGATION):** It should be understood that given a particular configuration of a neural network and a particular dataset, there exists a loss value for each and every possible combination of the neural network weights and biases. Thus there exists a loss surface.

   The loss surface of neural networks can get increasingly complex and non convex as the depth of the network increases. Despite this, the process of determining the gradient of the loss curve at a particular point and then taking a miniscule step in the negative direction of the gradient (ie: gradient descent), has proved to be effective in leading us to some point on the losssurface that is quite close to the absolute minima (ie: A good enough local minima). Shown below is a dimensionally reduced visualisation of a particular deep neural network's loss surface (ResNet-110):



The gradient of the loss surface at a particular point is computed using a technique called back propagation, that relies on the "chain rule" of partial derivatives which is described below:





Assuming that the loss and activation functions used are differentiable, we can derive the expressions

for the loss gradients with respect to each weight & bias in the neural network using the equations shown above (ie: Expressions for: **∂L/∂w** and **∂L/∂b** at any node/neuron, where **L** is the loss function).

Given that the current weight matrices and bias vectors of the neural network represent a particular location on the loss surface, the process of Back Propagation involves updating the weights & biases of each neuron by a miniscule amount, in the direction opposite to the loss gradients calculated for each of those neurons. This is akin to taking a very small step in the downward direction of maximum steepness from the present location.

5. **ITERATION FOR A SUITABLE NUMBER OF EPOCHS:** The steps previously described are just the first among a series of iterations that are aimed at finding a set of weights & biases for the neural network that would produce the least loss. After each iteration, another chunk of data of the same size is randomly selected and steps 2, 3 & 4 are repeated on it using the updated weights obtained from the previous iteration.

   The number of iterations should approximately be equal to the size of the whole dataset divided by the size of the randomly sampled chunk of data. The number of iterations thus computed represents one pass of the entire training dataset through the neural network. This is called an **Epoch**.

   The process described above is repeated for a suitable number of epochs (in the order of hundreds and above depending on data complexity and size) till the loss value generated by the neural network no longer shows any improvement. After each epoch the resulting neural network thus obtained is used to predict on the train data and the evaluation data and the performance of the network is recorded/monitored. The model corresponding to the epoch number that produces the best performance on both - the train and evaluation sets are then chosen. This model is then tested on the testset.

The **optimization** process described above is called **Stochastic Gradient Descent**. It is so called due to the fact that a randomly sampled chuck of data is used during each iteration instead of the whole dataset. Optimization in the case of neural networks, could be seen as a search process through a loss landscape, where the search objective is to find the location of lowest point (or at least some point close to the lowest point) on the loss surface.

ACTIVATION FUNCTIONS (CREATING NON LINEAR TRANSFORMATIONS OF INPUT DATA):

One of the main reasons for the effectiveness of neural networks is that it can approximate functions that represent a complex nonlinear mapping between the given inputs and the desired outputs. The nonlinearity in neural networks is provided by the activation functions used.

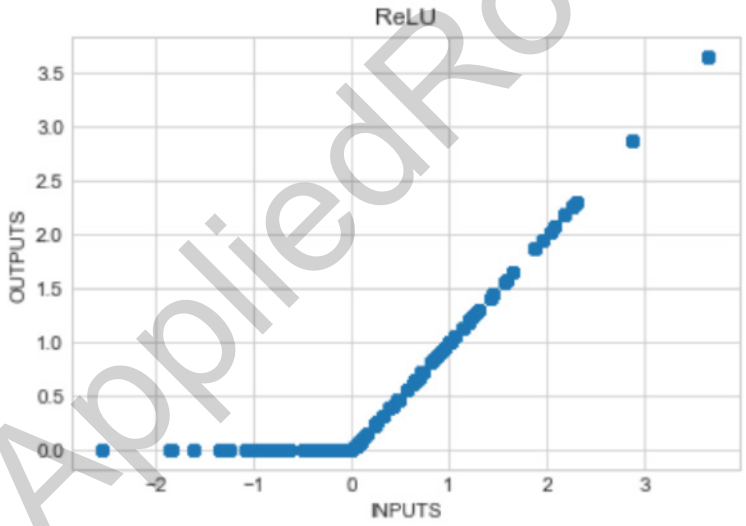
Linear transformation means that all of the output vectors that a layer produces, exist in a feature space whose axes are symmetric and straight. Non linearity means that the output vectors exist in a feature space whose axes are curved, irregular and/or asymmetric. A neural network layer would only be performing a simple linear transformation, if no activation function were used (ie: The transformation of a vector by a matrix).

In the earlier times the sigmoid or the tanh functions were used to create nonlinearity. But these proved to be cumbersome, as the depths used in neural networks increased in modern times. Phenomena such as the vanishing gradient problem made it hard to train deep neural networks. It was eventually discovered that using the ReLU function for activation, usually solved this problem. Most modern neural networks generally use the ReLU or some variant of it, as their activation function.

THE ReLU FUNCTION:

ReLU stands for Rectified Linear Unit. The definition of a ReLU is: **a = max(0, z)**. For neural networks **z** is the output at each neuron which is = **w*x + b**. The function creates a profile of the kind shown below. Its output is linear as long as its input value **z** is greater than zero, but for values less than zero, it produces a constant output of zero (nonlinear response).

```
x = list(np.random.randn(100))*10
y = [max(0, i) for i in x] #------- ReLU

fn_scatterplot(x, y)
```



Observing the slopes of the ReLU function tells us that the slope (or derivative) of the ReLU function for all values greater than one is always one and for values less than one, it is zero. At input = zero the slope of the ReLU is indeterminate.

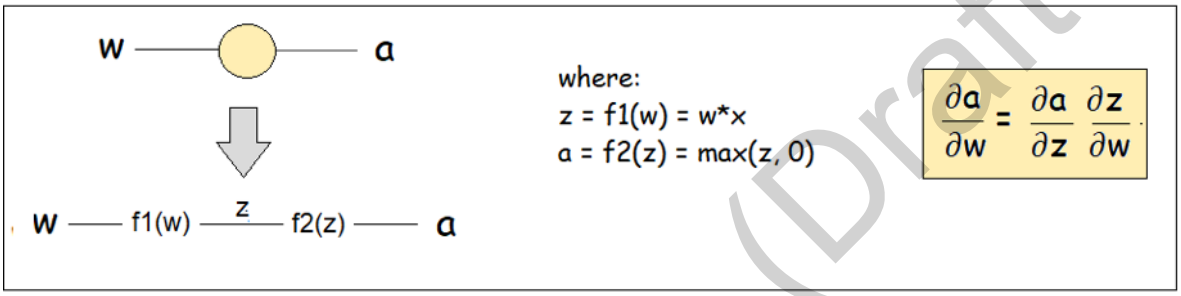
DERIVATIVE OF A NEURON'S OUTPUT WITH RESPECT TO ITS WEIGHT:

Consider the information path through a single neuron, its function can be decomposed into 2 sub functions acting in sequence, as shown below:

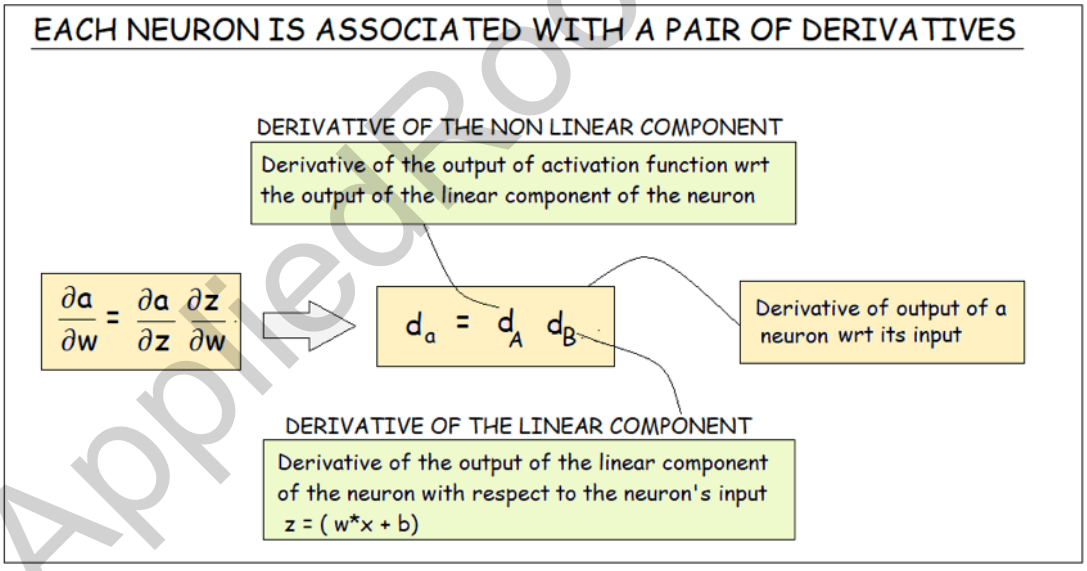
1. First the linear function: **z = f1(w) = w*x + b** and
2. Second the nonlinear activation function: **a = f2(z)**

Note how the weight **w** is considered as the input variable. In machine learning the entire data matrix **X** is seen as a single multidimensional constant, intrinsic to the loss function and the weights are considered as the variables we have to solve/optimize for.

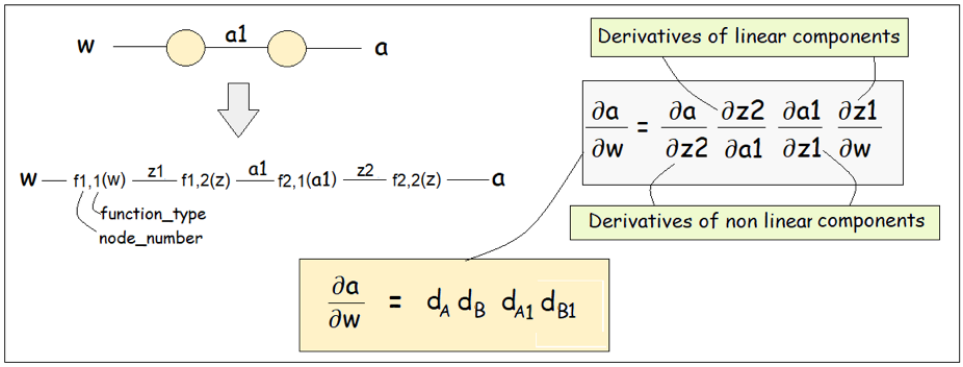
The derivative of the neuron's output '**a**' with respect to its weight **w** can be obtained using the chain rule as shown in the image below (We shall ignore the bias term to maintain simplicity):



Observing the above formulation it can be said that each neuron is associated with two fundamental derivatives:



Now consider a path between 2 sequential neurons. The derivative of the output at the second neuron, with respect of the weight of the first neuron, can be derived using the chain rule as shown below:



Note that the inputs of all neurons except of the first neuron in the path, is the output of the previous neuron. For the first neuron, its own weight is considered as its input.

GRADIENT OF LOSS WITH RESPECT TO WEIGHT (& BIAS) OF ANY NEURON:

Consider a loss function **L(y, ŷ)** where **ŷ = 'a'** the output of the neural path chosen. The derivative of the loss function with respect to the weight of any node in the neural path, can be found by using the derivative chain expression shown below.

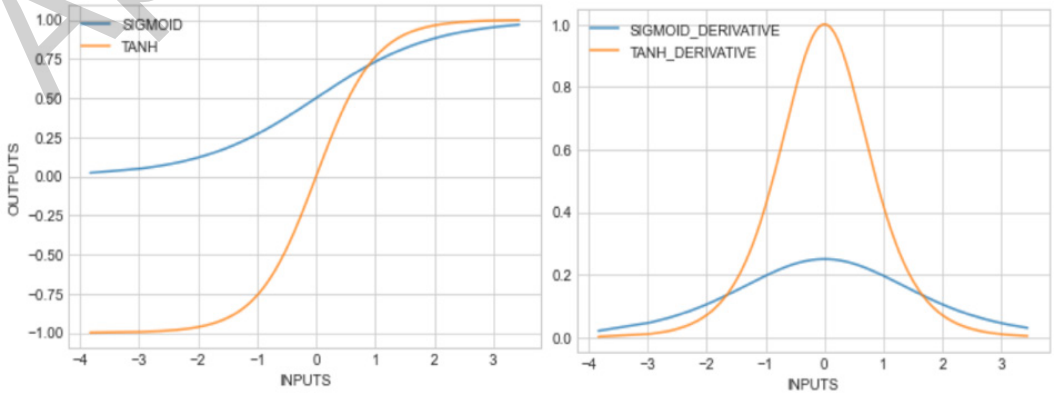
$$\frac{\partial L}{\partial w_n} = \frac{\partial L}{\partial a}\ \frac{\partial a}{\partial z}\frac{\partial z}{\partial a_1}\ \frac{\partial a_1}{\partial z_1}\frac{\partial z_1}{\partial a_2}\ \cdots\cdots\ \frac{\partial a_n}{\partial z_n}\frac{\partial z_n}{\partial w_n}$$

$$\frac{\partial L}{\partial w_n} = d_L\ d_A\, d_B\ d_{A1}\, d_{B1}\ \cdots\cdots\ d_{An}\, d_{Bn}$$

THE VANISHING GRADIENT PROBLEM:

Every neuron is associated with a chain of 2 derivatives and hence the **nth** neuron from the output neuron will have a derivative chain of length **2n**. Thus a neuron at the 100th layer from the output layer will have 200 derivatives in its derivative chain. This derivative chain is solely composed of 2 types of derivatives - derivatives corresponding to the nonlinear function($\mathbf{d_A}$) and those corresponding to the linear function ($\mathbf{d_B}$).

As mentioned earlier, before ReLU became the standard, the sigmoid and tanh functions were used as activation functions.. The image below shows the function and derivative curves of these functions.



As seen from the plots above, as the magnitude of the input value increases, the slope (and hence the derivatives) of both functions decrease and come very close to zero for magnitudes greater than 3. The maximum derivative value possible for a sigmoid function is 0.25 and for tanh function it is 1. The sigmoid function has all its derivative values less than 1 and the tanh function has most of its derivative values less than 1 (except when the input value is zero).

Given the facts that the derivatives for the sigmoid and tanh functions ($\mathbf{d_A}$ values) are for most part less than 1 and that the further away a neuron is from the output neuron, the longer the derivative chain associated with that neuron, it is quite probable that the derivative of the output with respect to a "far away" neuron's weight will become negligibly small and disappear (ex: For 3 layers: **0.001^3 = 10^-9**) This effect is further compounded if the $\mathbf{d_B}$ values also are lesser than one. If this happens across many neurons in a layer, the layer can become defunct, since it has lost its capacity to "learn". This is the vanishing gradient problem.

If the activation function used is the ReLU function, then the first type of derivative ($\mathbf{d_A}$) will have values of either zero or one. This greatly reduces the probability of the phenomenon of vanishing gradient simultaneously happening to many neurons in a layer. Even though the derivative of the ReLu function at input = zero is indeterminate, the function performs surprisingly well as an activation function.

THE EXPLODING GRADIENT PROBLEM:

This is opposite of the disappearing gradient problem. This is again due to formation of long derivative chains for neurons that are further away from the output layer. The accumulated gradient for these neurons can get quite large and even cause overflow or NaN values, if the values of all the derivative components (ie: all $\mathbf{d_A}$ and $\mathbf{d_B}$) are greater than or equal to one (ex: For 50 layers: 2^50 = 1.1 x 10^15). These large gradients result in large updates to the weights of these "far away" neurons, resulting in unstable networks.

Exploding gradients are relatively rare and can be tackled using gradient clipping techniques or by applying batch normalization or even by using the proper weight initialization technique. Details of these techniques will be discussed in the "Deep Learning" chapter further on in the book.

COMPONENTS OF NEURAL NETWORK DESIGN :

1. **NETWORK CONFIGURATION:** This involves making decisions about:
   a. The number of hidden layers and
   a. The number of neurons in each of these individual layers (layer widths).

   There are no fixed rules for network configuration except that the number of parameters used (weights & biases) should be appropriate for the size and complexity of the data. Deep learning models are generally designed such that they are over parameterized. One could use the multiclass softmax logistic regression classifier as a reference for the minimum

number of neurons needed. The multiclass logistic regression model uses the same number of linearly activated neurons as the dimensionality of the dataset. It is advisable to configure the neural network such that the total number of neurons it contains within its hidden layers are at least 50% more than the number of features in the dataset.

2. **WEIGHT INITIALIZATION:** This involves using an optimal weight initialization strategy to start off the gradient descent along the loss surface. The following are some of the initializations used in modern neural networks:
   a. Xavier-Glorot Normal Initialization
   b. Xavier-Glorot Uniform Initialization
   c. He Normal Initialization
   d. He uniform Initialization

   Details of these will be discussed in the "Deep Learning" chapter further on in the book. For now we will be using the default weight initialization provided to us by the Keras module.
3. **LOSS FUNCTION :** This involves choosing the right loss function for the task at hand.
   a. Squared loss (regression)
   b. Log loss or Binary cross entropy (binary classification)
   c. Cross entropy loss (multiclass classification)
4. **OPTIMIZATION ALGORITHM:** The Stochastic Gradient Descent (SGD) algorithm is the most basic of all the gradient descent algorithms implemented. It is quite effective most of the time. It often gives results as good as those of the more advanced methods, except that it may be significantly slower. The advanced algorithms use momentum (weighted sum of previous gradients), adaptive learning rates and other methods to avoid getting trapped at local minima. Some of these methods are mentioned below:
   a. SGD with momentum
   b. Adagrad
   c. RMS prop
   d. Adam

   Details of these will be discussed in the "Deep Learning" chapter further on in the book.
5. **BATCH SIZE:** This refers to the size of the data chunk that is stochastically sampled at every iteration. There is a tradeoff between the batch size used and the stability of the learning process in deep learning. This is because:
   a. The updates of the weights and biases are made using the loss gradients computed with respect to the current data chunk.
   b. The more larger a sample is, the more representative it is of the parent dataset it was sampled from.

The smaller the batch size, the more variance there is going to be between the various data chucks that are stochastically sampled. This results in the model learning a more zig zag path along the loss surface.

There also a regularization effect that is produced by noisiness of the learning path, it randomly covers more "Area" on the loss surface and hence has lesser chances to overfit to the current chunk of data that the model is learning from.

There is also the constraint of computational resources. The larger the batch size, the more the computational capacity required at every iteration. The batch size chosen should be such that it is appropriate for the computational resources at hand.

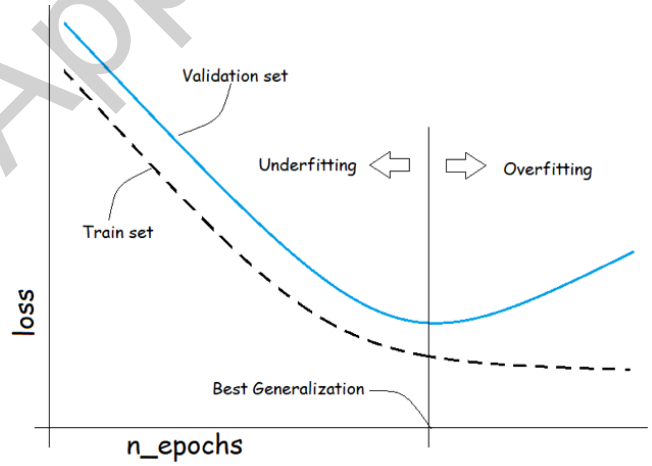
In general, batch sizes of 32, 64 or 100 work quite well for most datasets. The appropriate batch size should be decided based on understanding the trade offs between the various elements mentioned above.

6. **LEARNING RATE:** This refers to the size of the step taken by the model, along the gradients computed per iteration. The smaller the learning rate, the more the ability of the model to navigate through the complexity of the loss curve, but comes at the cost of increase in time required to train the model.

   Bigger step sizes means more probability of error propagation. In other words, if the gradient computed at the present iterations is not optimal, then taking a bigger step would mean a larger movement in the wrong direction.

   Learning rates in the range of 0.0001 to 0.1 are generally used in most deep learning models. The learning rate is a parameter of the "optimizer" (optimization algorithm) used.

7. **NUMBER OF EPOCHS:** An epoch is the number of iterations required for one pass of the entire training dataset through the neural network. Instead of using randomly sampled batches, imagine the process of doing gradient descent using the whole dataset at every iteration. One epoch of iterations, using randomly sampled batches of data would be approximately equivalent to one iteration of gradient descent using the whole data. The thumb rule to estimate a suitable epoch size, is to find how many repetitions of the batch size produces the training data size.



It is not uncommon to train neural network models for hundreds of epochs. The image above is a simplified illustration of a loss performance curve over multiple epochs. The performance of the neural network is logged/monitored during training and the weights and biases of the model at the epoch that gives the best generalization is recalled and used for further prediction.

STEPS FOR IMPLEMENTING NEURAL NETWORK MODELS (BINARY CLASSIFICATION EXAMPLE):

We will be using the "**Sequential**" class from the Keras Deep Learning library for creating the neural network. This class provides us a simple API for sequentially declaring the configurations of the neural network layers we intend to use.

We will use the **Credit Card Fraud** dataset for demonstration purposes. Since we have discussed the binary classification pipeline in quite some detail in the previous chapters, we will pick up from the "model training and evaluation" stage in the pipeline.

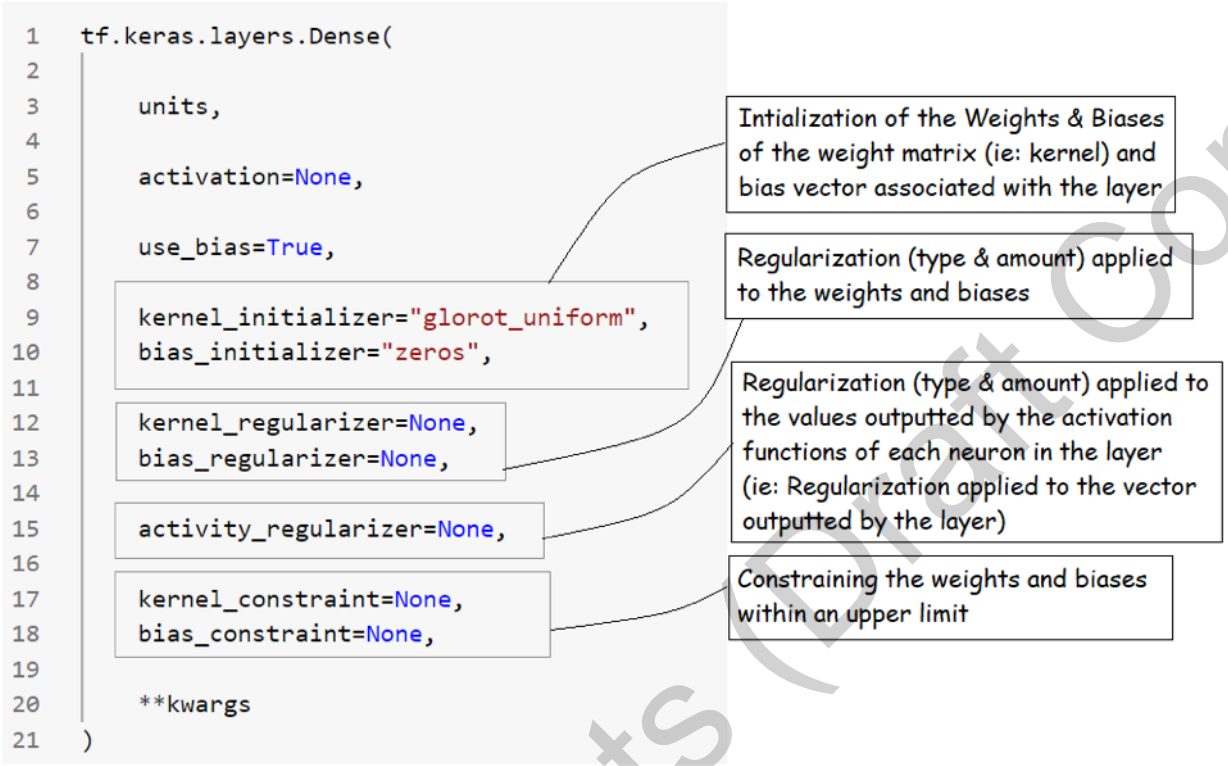
1. MODEL CONFIGURATION:

FIrst we configure/design the neural network. Say we choose to have 2 hidden layers. The first hidden layer we decide to have ten neurons and in the second we choose to have five. Since it is a binary classification problem, we choose to have only one neuron in the output layer with a sigmoid activation function. The code below creates a neural network model of configuration just described.

```
import tensorflow as tf
from tensorflow import keras

model_1 = keras.Sequential(name = 'NN_1')

model_1.add(keras.Input(shape=(n_feats,)))
model_1.add(keras.layers.Dense(10, activation="relu"))
model_1.add(keras.layers.Dense(5,  activation="relu"))
model_1.add(keras.layers.Dense(1,  activation="sigmoid"))

model_1.summary()
```

```
Model: "NN_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 10)                160
_________________________________________________________________
dense_1 (Dense)              (None, 5)                 55
_________________________________________________________________
dense_2 (Dense)              (None, 1)                 6
=================================================================
Total params: 221
Trainable params: 221
Non-trainable params: 0
```

As can be seen from the output above, the model contains 221 trainable parameters. The "**keras.layer.Dense**" class basically represents hidden layers. The default arguments and keywords of this class is shown below:



As can be seen in the image above, given the default settings of a layer, the only specifications required are the number of neurons in the layer (ie: units) and the activation function. For most cases the default setting works quite well and hence will be using the same in the examples in this chapter.

2. MODEL COMPILATION:

This step involves defining:

1. The optimization technique/algorithm to use.
2. The loss function used for gradient descent purposes.
3. The performance metrics to be computed at each epoch.

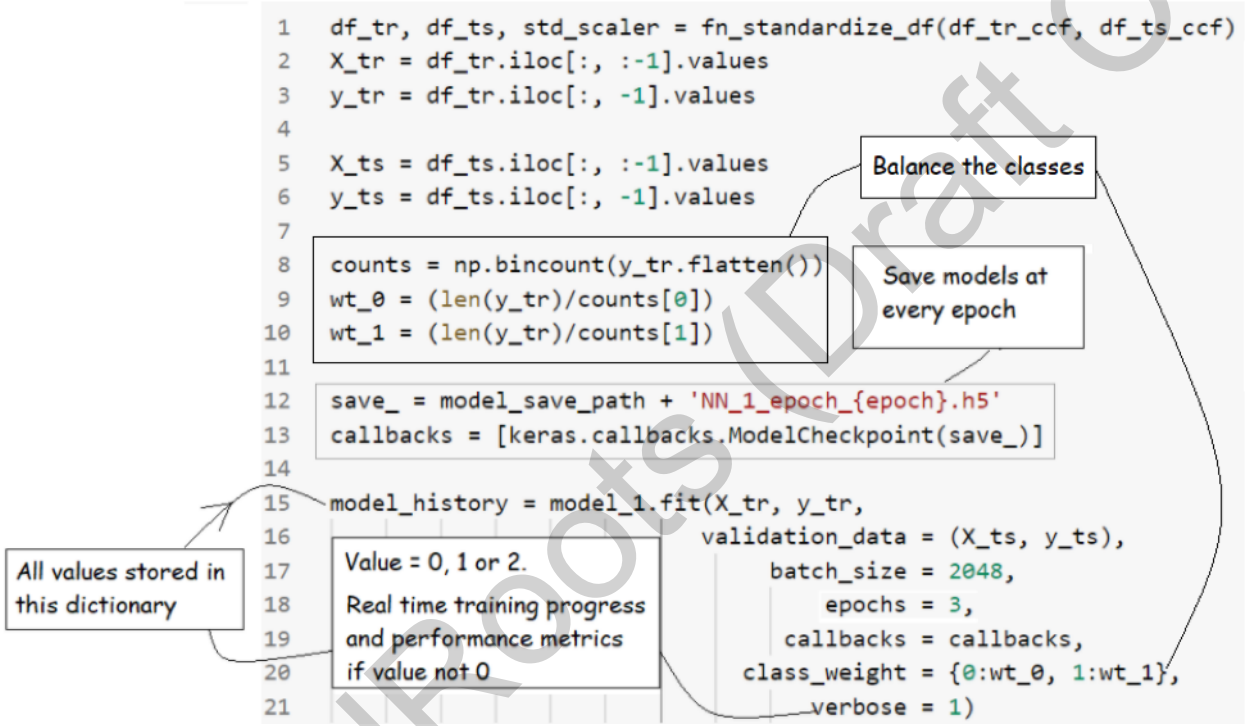
The Keras framework provides a comprehensive set of options for each one of these requirements. Shown below is the code to compile the Neural Network Model ('**NN_1**') just configured earlier.

```
lr = 1e-2  #---- Learning Rate
optimizer=keras.optimizers.SGD(lr)

metrics =  [keras.metrics.BinaryAccuracy(name='acc')]

model_1.compile(optimizer=optimizer, loss="binary_crossentropy", metrics=metrics)
```

The code shown above will make Keras compute and store the metrics specified (Accuracy), for both the train set and evaluation set, at each epoch, while training the model. The metrics are stored in a dictionary. In addition to these metrics, the loss values at each epoch are also stored in the same dictionary.

3. MODEL FITTING:

After we configure and compile the neural network, we train the model using the code shown below. The code trains the model for 5 epochs and saves the model (weights & biases) arrived at at each epoch in the location specified.




```
Epoch 1/3
105/105 [==============] - 0s 4ms/step - loss: 0.8304 - acc: 0.7233 - val_loss: 0.4175 - val_acc: 0.9143
Epoch 2/3
105/105 [==============] - 0s 3ms/step - loss: 0.5361 - acc: 0.9497 - val_loss: 0.2669 - val_acc: 0.9596
Epoch 3/3
105/105 [==============] - 0s 3ms/step - loss: 0.4336 - acc: 0.9667 - val_loss: 0.1949 - val_acc: 0.9710
```

As can be seen from above, the neural network "NN_1" is trained over 3 epochs. The training progress and performance metrics specified are displayed in real time as and when the model is being trained. The performance metric chosen is accuracy - this is just for demonstration purposes. As discussed earlier accuracy is not a valid metric when dealing with highly imbalanced datasets. We can specify as many metrics as required and keras will compute and store them in the dictionary "**model_history**" as shown above. This dictionary can easily be inspected as shown below:

```
pd.DataFrame(model_history.history)
```

	loss	acc	val_loss	val_acc
0	1.040477	0.491408	0.546803	0.810646
1	0.575409	0.920406	0.312911	0.959570
2	0.434687	0.971855	0.217602	0.975176
3	0.380022	0.976767	0.172226	0.978723
4	0.353838	0.979236	0.159307	0.975808

The above discussion about model configuration, compiling and training describes the basic technique involved while using neural network based models. We will now approach the same problem (credit card fraud detection) in a more formal manner. All the helper functions used previously for binary classification tasks can be reused for neural network models. One extra helper function "**fn_NN_binary_clf**" has been created, designed specifically to be useful for imbalance binary classification tasks using neural networks. The details of this function is discussed below:

df_performance_model = fn_NN_binary_clf(model, optimizer, df_tr, df_ts, model_save_path, batch_size = batch_size, epochs = epochs, class_weights = class_weights)

Given the input arguments and keyword arguments, the function trains the model over the number of epochs specified, saving the models obtained after each epoch and after completing all epochs, it returns the data frame **df_perform**, which is a record of the binary classification performances of the models at each epoch (precision and recall for both classes). The metrics specified inside this function is the true positive, false positive, true negative and false negative counts at each epoch. From these metrics, precision and recall are derived and returned in **df_perform** shown below:

```
model = model_1
lr = 1e-2
optimizer=keras.optimizers.SGD(lr)
df_tr_, df_ts_, std_scaler = fn_standardize_df(df_tr_ccf, df_ts_ccf)

model_save_path =  data_path + 'NN_MODELS/model_1/'
batch_size, epochs = 2048, 250
class_weight = {0:0.11, 1:99.89}

z = fn_NN_binary_clf(model, optimizer, df_tr_, df_ts_, model_save_path,
                     batch_size=batch_size, epochs=epochs, class_weight = class_weight)
df_perform_1, df_history_1 = z
```

Balancing class weights :
Original data distribution:
{0: 99.82, 1: 0.18}

100% 250/250

```
regd_columns = 'tr_prec_1 tr_rec_1 tr_prec_0 tr_rec_0 diff_rec_1 diff_rec_0'.split()

df_perform_1.loc[:, regd_columns].describe()
```

	tr_prec_1	tr_rec_1	tr_prec_0	tr_rec_0	diff_rec_1	diff_rec_0
count	250.000000	250.000000	250.000000	250.000000	250.000000	250.000000
mean	0.033427	0.931096	0.999865	0.947347	0.006748	0.003228
std	0.005279	0.014566	0.000033	0.019997	0.009963	0.002044
min	0.008842	0.763959	0.999497	0.818972	0.000311	0.000240
25%	0.034222	0.926396	0.999858	0.951364	0.004714	0.002608
50%	0.035041	0.934010	0.999872	0.952384	0.005439	0.003228
75%	0.035637	0.936548	0.999877	0.953229	0.006785	0.003650
max	0.036794	0.946701	0.999897	0.955374	0.144204	0.031492

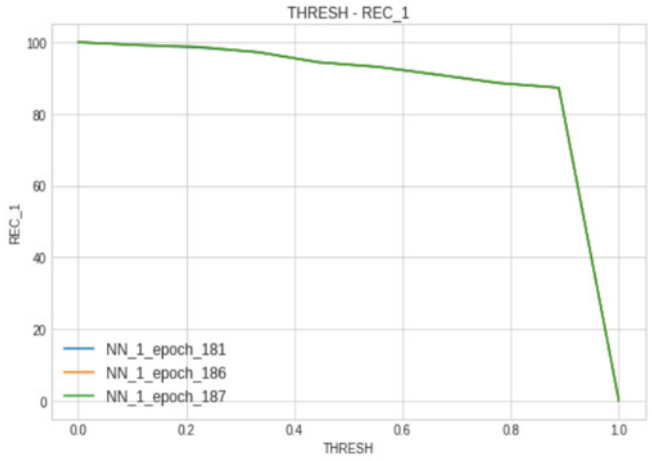
Note that the model has been trained over **250 epochs** and the table above gives us the statistical summary of the performance metrics achieved during those epochs. We then filter out the best performing models using the code shown below:

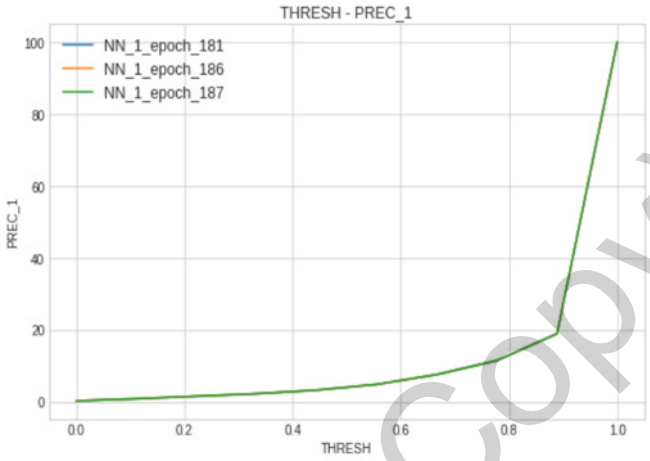
```
dff = df_perform_1.loc[:, regd_columns]
df1 = dff[dff.diff_rec_1 < 0.0025]
df2 = df1[df1.tr_rec_1 > 0.936]
df_filtered = df2[df2.tr_rec_0 > 0.958]
df_filtered
```

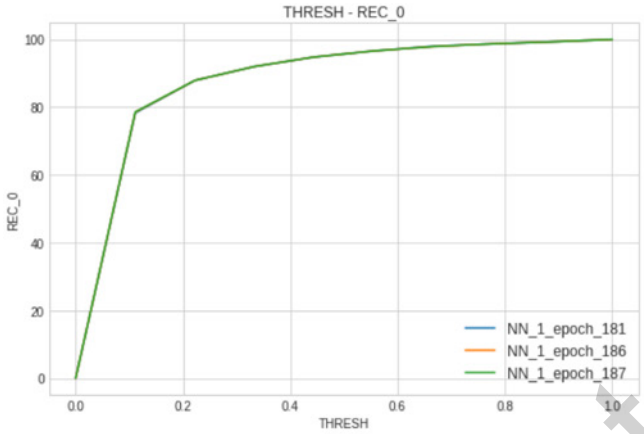
	tr_prec_1	tr_rec_1	tr_prec_0	tr_rec_0	diff_rec_1	diff_rec_0
NN_1_epoch_181	0.039546	0.936548	0.999878	0.958029	0.002227	0.002996
NN_1_epoch_186	0.039665	0.936548	0.999878	0.958160	0.002227	0.003321
NN_1_epoch_187	0.039776	0.936548	0.999878	0.958282	0.002227	0.003654

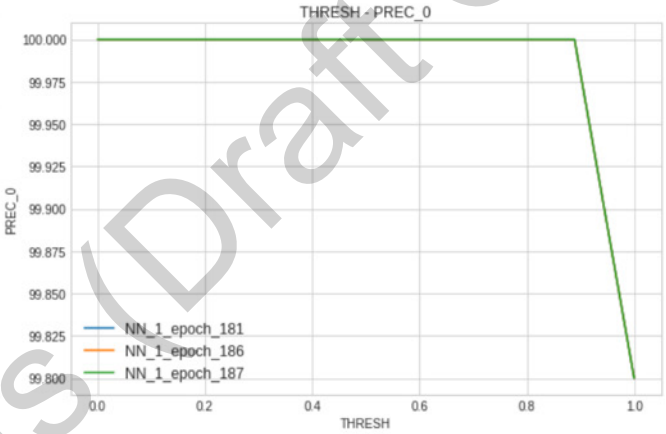
We then inspect the Precision-Recall curves for the three models shown above to make a decision as to which model to choose for our task.

```
def fn_load(model_name): return keras.models.load_model(model_save_path + model_name +'.h5')

model_save_path = data_path + 'NN_MODELS/model_1/'
list0_model_names = list(df_filtered.index)
list0_models = [fn_load(i) for i in list0_model_names]

df_Xy_ = df_tr_
legend = list0_model_names

fn_performance_models_data(list0_models, df_Xy_, legend, NN=True)
```



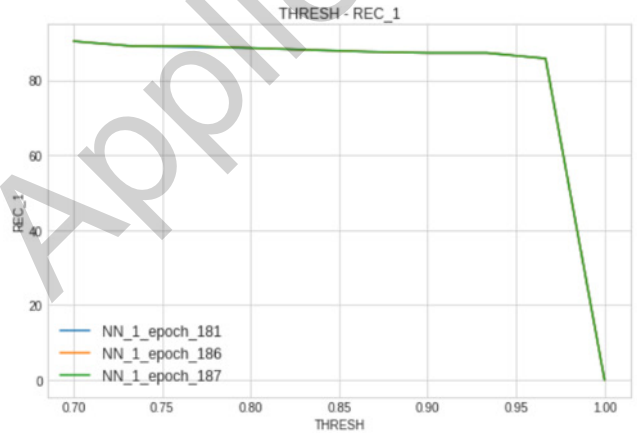


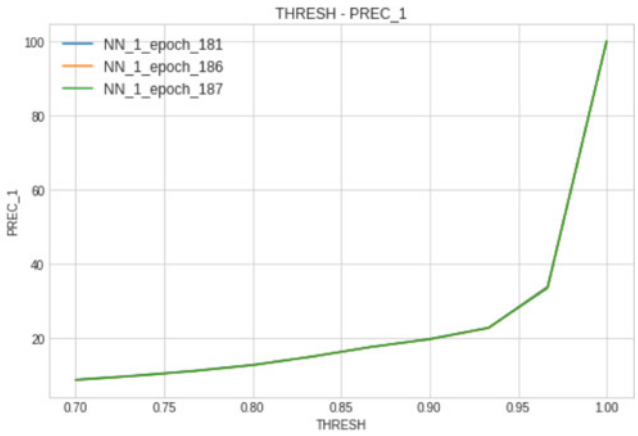


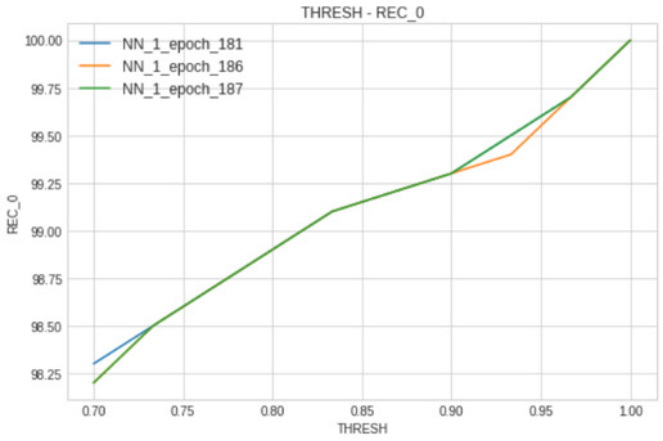


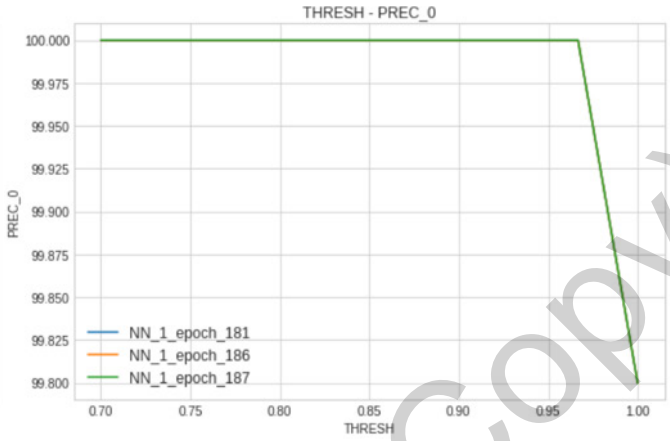
We can further zoom into the threshold range of 0.7 - 1.0 to get a clearer picture of the model's performance, as shown below:

```
listO_thresholds = np.linspace(0.7, 1, 10)
legend = listO_model_names

fn_performance_models_data(listO_models, df_Xy_, legend,
                           listO_thresh_cls_1 = listO_thresholds, NN=True)
```









Studying the plots shown above we choose the model "**NN_1_epoch_187**" thresholded at 0.96. We then check how this model generalizes over the trainset and the testset as shown below:

```
df_Xy_ = df_tr_
model_ =  listO_models[2]  #--NN_1_epoch_187
threshold_class_1 = 0.96

fn_test_model_binary_clf(df_Xy_, model_, threshold_class_1 = threshold_class_1)
```

```
--------------------
LOGLOSS : 0.1395
ACCURACY: 99.584
--------------------
```

	prec	rec
class_0	100.0	99.6
class_1	28.9	86.3

```
df_Xy_ = df_ts_

fn_test_model_binary_clf(df_Xy_, model_, threshold_class_1 = threshold_class_1)
```

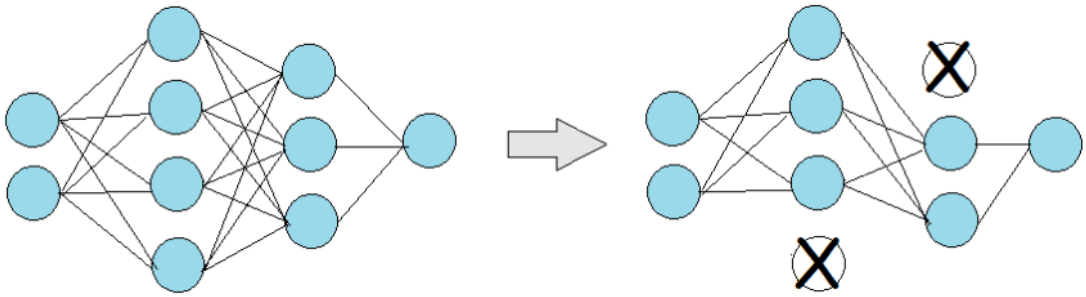
```
--------------------
LOGLOSS : 0.1484
ACCURACY: 99.489
--------------------
```

	prec	rec
class_0	100.0	99.5
class_1	23.7	88.8

As can be seen from above, the model generalizes quite well to the testset and gives a much better performance than the Logistic Regression model.

DROPOUT :

Dropout is a simple and surprisingly effective neural network regularization technique. It involves stochastically dropping out (make value = 0) the output values of the neurons of a layer (or set of layers), at every iteration during the gradient descent optimization process.



The “dropping out” is done based on a probability applied across all neurons in the layer. This probability, called the **dropout rate**, is a hyperparameter that has to be chosen/tuned by the user. Therefore for a layer subjected to dropout regularization with 30% dropout rate, every neuron in that layer has a 30% chance of being dropped out. This results in a different configuration of that layer at every iteration. In other words, the neural network is composed of a different subset of neurons at every iteration, as shown in the image above. Dropouts can be applied independently on multiple layers, with different dropout rates for each layer.

When a particular layer in the neural network is subjected to dropout regularization, the neurons in the layer subsequent to that layer are forced to learn their weights and biases such that, as a whole, the network performs as well as possible despite the dropout imposed. This creates a regularization effect.

Using dropout regularization increases the number of epochs (ie: iterations) required to reach convergence, however the training time required for each epoch is lesser.

THE NEED FOR SCALING WHEN USING DROPOUT:

A neural network whose layers are subjected to dropout while training, will still use all of its neurons during testing or real world prediction. For example, if the dropout probability is “**p**”, then on an average **p** percent of the neurons will be shut down and only the remaining “**q**” percent of the neurons are contributing to the prediction (where: **q = 1 - p**, is called the **keep probability**). Therefore during the testing phase, to emulate the same behavior of the network as in the training phase, the activations of the layers subjected to dropout are scaled (multiplied) by a factor of **q**. This kind of dropout is called **direct dropout**. This kind of dropout requires the extra step of scaling the network during the testing phase.

Another more preferred approach is **inverted dropout**, where the required scaling adjustment is done during the training phase itself. In this approach, the activations of the neurons in a layer subjected to dropout regularization, are scaled by the inverse of the keep probability (ie: 1/q). Inverted dropout is how dropout is implemented in most deep learning frameworks.

THE DROPOUT LAYER:

To apply dropout regularization to a particular layer we create a dropout layer after it and specify the dropout probability desired. This is done using the keras.layers.Dropout class. Dropout layers created using this class perform inverted dropout on the layer preceding it.

One of the basic techniques for creating high performance neural network models with good generalization, is to:

1. Configure highly over parameterized models that produce close to 100% performance on the train set.
2. Then use dropout layers and tune their dropout probability till we achieve the required generalization with respect to the test set.

The code in the image below implements this strategy by creating an over parametrized model (around 12700 trainable parameters) for the same example as before and uses a dropout of 20% between the hidden layers.

```
model_2 = keras.Sequential(name = 'NN_2')

model_2.add(keras.layers.Dense(30, activation="relu", input_shape=(n_feats,)))
model_2.add(keras.layers.Dropout(0.2))
model_2.add(keras.layers.Dense(50, activation="relu"))
model_2.add(keras.layers.Dropout(0.2))
model_2.add(keras.layers.Dense(100, activation="relu"))
model_2.add(keras.layers.Dropout(0.2))
model_2.add(keras.layers.Dense(50, activation="relu"))
model_2.add(keras.layers.Dropout(0.2))
model_2.add(keras.layers.Dense(10, activation="relu"))
model_2.add(keras.layers.Dense(1, activation="sigmoid"))

model_2.summary()
```

```
Model: "NN_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_3 (Dense)              (None, 30)                480
_________________________________________________________________
dropout (Dropout)            (None, 30)                0
_________________________________________________________________
dense_4 (Dense)              (None, 50)                1550
_________________________________________________________________
dropout_1 (Dropout)          (None, 50)                0
_________________________________________________________________
dense_5 (Dense)              (None, 100)               5100
_________________________________________________________________
dropout_2 (Dropout)          (None, 100)               0
_________________________________________________________________
dense_6 (Dense)              (None, 50)                5050
_________________________________________________________________
dropout_3 (Dropout)          (None, 50)                0
_________________________________________________________________
dense_7 (Dense)              (None, 10)                510
_________________________________________________________________
dense_8 (Dense)              (None, 1)                 11
=================================================================
Total params: 12,701
Trainable params: 12,701
Non-trainable params: 0
_________________________________________________________________
```

We then train the configured model using the function "**fn_NN_binary_clf**" just as before:

```
model = model_2
lr = 1e-2
optimizer=keras.optimizers.SGD(lr)

model_save_path =  data_path + 'NN_MODELS/model_2/'

batch_size, epochs = 2048, 200
class_weight = {0:0.11, 1:99.89}

z = fn_NN_binary_clf(model, optimizer, df_tr_, df_ts_, model_save_path,
                     batch_size=batch_size, epochs=epochs, class_weight = class_weight)
df_perform_2, df_history_2 = z
```

100% 200/200

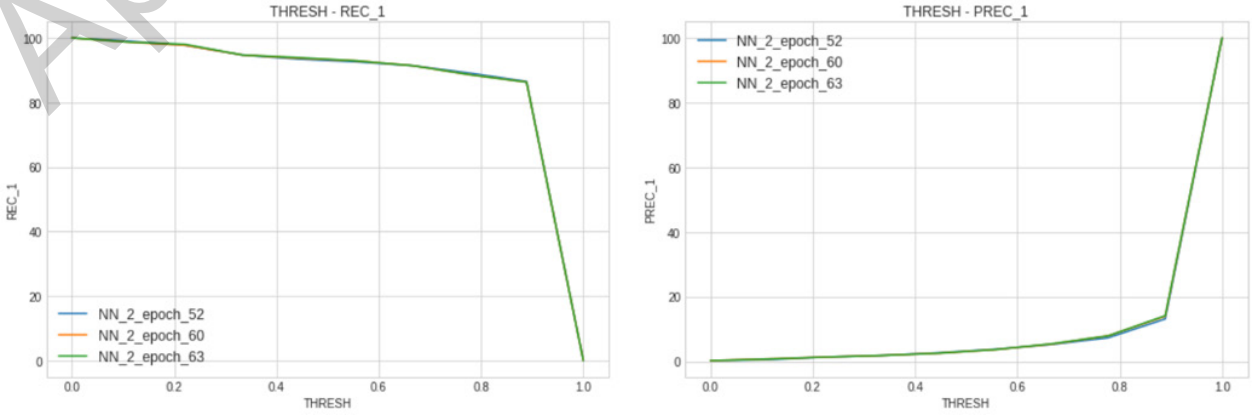
We then filter the best performing models using the code shown below:

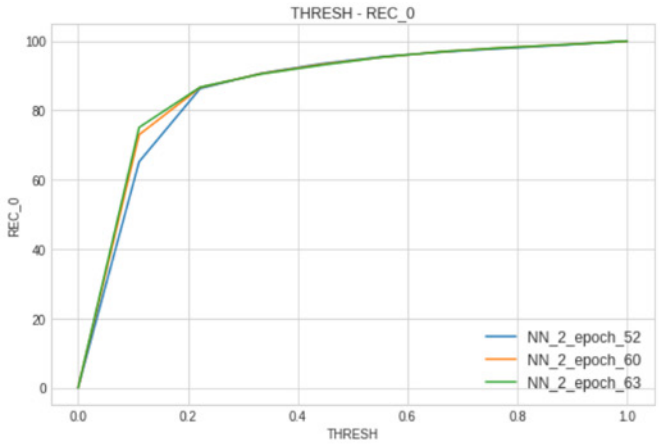
```
dff = df_perform_2.loc[:, regd_columns]
df1 = dff[dff.diff_rec_1 < 0.05]
df2 = df1[df1.tr_rec_1 > 0.93]
df_filtered_2 = df2[df2.tr_rec_0 > 0.94]
df_filtered_2
```

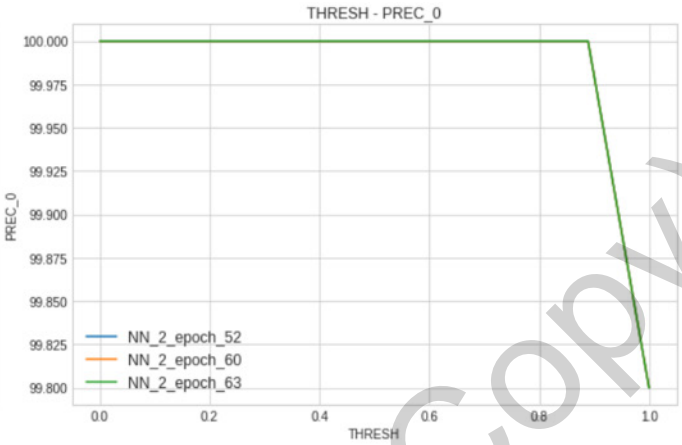
	tr_prec_1	tr_rec_1	tr_prec_0	tr_rec_0	diff_rec_1	diff_rec_0
NN_2_epoch_52	0.029532	0.931472	0.999866	0.943521	0.007303	0.000516
NN_2_epoch_60	0.029504	0.931472	0.999866	0.943465	0.007303	0.002395
NN_2_epoch_63	0.028417	0.931472	0.999866	0.941235	0.007303	0.000749

We then check the precision - recall performance of the filtered/selected models as shown below:

```
listO_model_names = list(df_filtered_2.index)
listO_models = [fn_load(i) for i in listO_model_names]

df_Xy_ = df_tr_
legend = listO_model_names

%time fn_performance_models_data(listO_models, df_Xy_, legend, NN=True)
```

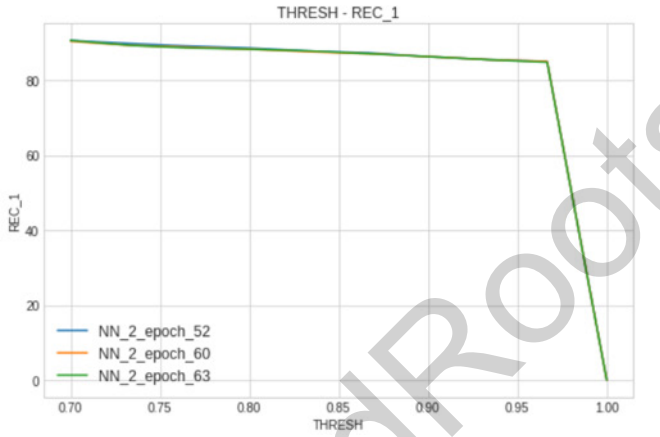


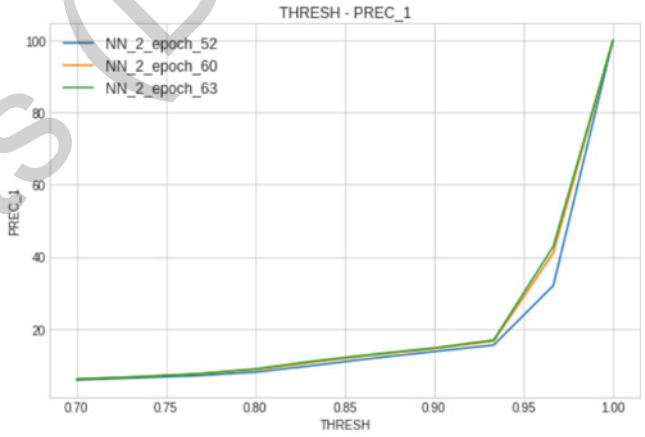


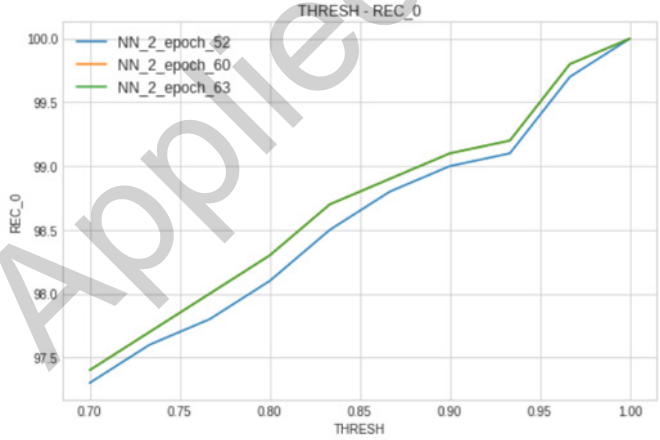


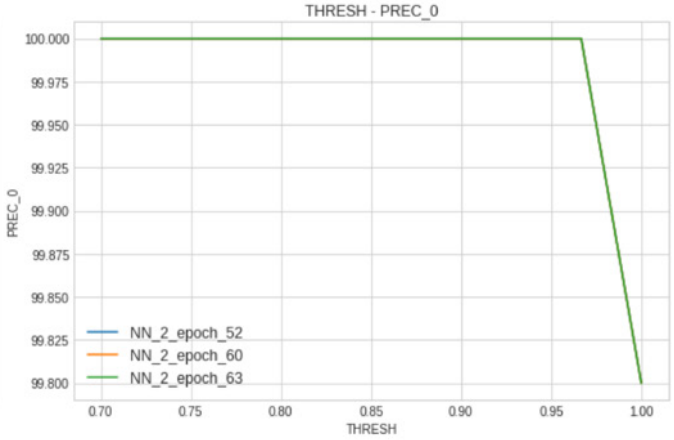
Zooming into the 0.4 to 0.85 threshold range, we have:

```
listO_thresholds = np.linspace(0.7, 1, 10)
legend = listO_model_names

fn_performance_models_data(listO_models, df_Xy_, legend,
                           listO_thresh_cls_1 = listO_thresholds, NN=True)
```









Choosing model_63, thresholded at 0.96, we check for performance across train and test sets:

```
df_Xy_ = df_tr_
model_ =  list0_models[2]  #--NN_1_epoch_63
threshold_class_1 = 0.96

fn_test_model_binary_clf(df_Xy_, model_, threshold_class_1 = threshold_class_1)
```

```
--------------------
LOGLOSS : 0.1896
ACCURACY: 99.657
--------------------
```

	prec	rec
class_0	100.0	99.7
class_1	33.2	85.0

```
df_Xy_ = df_ts_

fn_test_model_binary_clf(df_Xy_, model_, threshold_class_1 = threshold_class_1)
```

```
--------------------
LOGLOSS : 0.1965
ACCURACY: 99.589
--------------------
```

	prec	rec
class_0	100.0	99.6
class_1	28.1	88.8

As can be seen from the performances of the model on the train and test sets above, the models gives a precision performance of 28% on the minority class which is the highest as yet while maintaining a recall performance comparable to all the previous models as yet.

TRADE OFFS WHEN DEALING WITH HIGHLY IMBALANCED BINARY CLASSIFICATION:

When dealing with highly imbalanced datasets like the credit card fraud dataset it is important not to make direct comparisons using percentage values. For instance, given a ratio of around 200,000 : 400 negative points to positive points, just 1 percent of the majority class being misclassified means 2000 misclassified points, which totally eclipses the minority class. Similarly 10 percent of the minority class being misclassified means just 40 misclassified points.

In the case of tasks such as the credit card fraud detection task one has to weigh how much recall we require on the “frauds” and how much lack of precision (amount of non fraud data points classified as fraud) we are willing to accept. With reference to the results achieved above, it means that the model is able to recover 356 frauds out of a total of around 400 frauds, but at the same time these 356 frauds are still mixed up with 2000 non fraud data points. Hence the low precision performance for the minority class.

For proper model evaluation one has to set an acceptable lower threshold on the recall of the majority class. For example we could say that the model should at least give 99 percent recall on the majority class and given this constraint, achieve the best recall (and precision) possible on the minority class.

NEURAL NETWORKS & GPUS:

Since neural networks are generally highly over parameterized, they are quite compute intensive and take much longer time to train. Graphical Processing Units (GPUs) are computational units previously used for computer animation purposes, which required a huge number of simple computations to be parallely processed. This attribute is ideal for Neural Network computations, since most of the computations in a neural network are trivially parallelizable.

The code examples in this book were implemented on Google Collaboratory, which is an free online GPU resource, designed specifically for Neural network implementation purposes.

MULTICLASS CLASSIFICATION USING NEURAL NETWORKS:

When using neural networks for multiclass classification, most of the steps stay the same as those described in the case of binary classification. One just has to configure a neural network work suitable for the specific task at hand (ie: the output layer has to have the same number of neurons as the number of unique labels in the dataset) and choose the appropriate loss function suitable for the task (ie: cross entropy loss). We will use the Penguin dataset as an example and start from the model training and evaluation stage.

MODEL CONFIGURATION:

As can be seen on the image below, we choose to have 6 hidden layers of various widths, with dropout layers having 20% dropout probability. Just as before we choose to have relu activations for each of the hidden layers and softmax activation for the output layer. This results in a model with 18613 trainable parameters. Note that this is an order of magnitude times larger than the number of datapoints we will be training on, which is just 321. Such over parameterization is normal for neural networks. We avoid the potential overfitting that can happen due to the over parameterization, by the use of dropout.

```
multi_class_model_1 = keras.Sequential(name = 'multi_NN_1')

multi_class_model_1.add(keras.Input(shape=(n_feats,)))
multi_class_model_1.add(keras.layers.Dense(20, activation="relu"))
multi_class_model_1.add(keras.layers.Dropout(0.2))
multi_class_model_1.add(keras.layers.Dense(100, activation="relu"))
multi_class_model_1.add(keras.layers.Dropout(0.2))
multi_class_model_1.add(keras.layers.Dense(100, activation="relu"))
multi_class_model_1.add(keras.layers.Dropout(0.2))
multi_class_model_1.add(keras.layers.Dense(50, activation="relu"))
multi_class_model_1.add(keras.layers.Dropout(0.2))
multi_class_model_1.add(keras.layers.Dense(20, activation="relu"))
multi_class_model_1.add(keras.layers.Dropout(0.2))
multi_class_model_1.add(keras.layers.Dense(10, activation="relu"))
multi_class_model_1.add(keras.layers.Dense(num_classes, activation="softmax"))

multi_class_model_1.summary()
```

```
Model: "multi_NN_1"
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_4 (Dense)              (None, 20)                100
_________________________________________________________________
dropout_2 (Dropout)          (None, 20)                0
_________________________________________________________________
dense_5 (Dense)              (None, 100)               2100
_________________________________________________________________
dropout_3 (Dropout)          (None, 100)               0
_________________________________________________________________
dense_6 (Dense)              (None, 100)               10100
_________________________________________________________________
dropout_4 (Dropout)          (None, 100)               0
_________________________________________________________________
dense_7 (Dense)              (None, 50)                5050
_________________________________________________________________
dropout_5 (Dropout)          (None, 50)                0
_________________________________________________________________
dense_8 (Dense)              (None, 20)                1020
_________________________________________________________________
dropout_6 (Dropout)          (None, 20)                0
_________________________________________________________________
dense_9 (Dense)              (None, 10)                210
_________________________________________________________________
dense_10 (Dense)             (None, 3)                 33
=================================================================
Total params: 18,613
Trainable params: 18,613
Non-trainable params: 0
_________________________________________________________________
```

MODEL TRAINING AND EVALUATION:

We use the function "**fn_NN_multi_clf**" to train the model and record/log its performance at each epoch. As shown below, we use a batch size of 32 and train the model for 250 epochs. It should be noted that Keras requires that classification labels have to be **label encoded** (ie: If the labels are strings, then each label has to be represented by a unique integer). We use Scikit Learn's label encoder for this purpose. We fit the encoder object to the labels of the train set and then transform both the train and test set using it. We can use the encoder object to inverse transform the integer labels back to string labels as and when required. The data used in the code below has been label encoded.

```
data_path = 'gdrive/My Drive/3A_PGD/CODE_PGD/CODE_ANN_basics/DATA/'

df_tr = pd.read_csv(data_path + 'df_tr_PENGUINS.csv')
df_ts = pd.read_csv(data_path + 'df_ts_PENGUINS.csv')

df_tr.shape, df_ts.shape
```

```
((252, 5), (69, 5))
```

```
from sklearn.preprocessing import LabelEncoder

labelencoder = LabelEncoder().fit(df_tr.labels)
y_tr = labelencoder.transform(df_tr.labels)
y_ts = labelencoder.transform(df_ts.labels)

df_tr = df_tr.iloc[:,:-1].assign(labels = y_tr)
df_ts = df_ts.iloc[:,:-1].assign(labels = y_ts)
```

We compute the weights required the imbalance in the classes as shown below:

```
df_tr.labels.value_counts()
```

```
0    115
2     93
1     44
Name: labels, dtype: int64
```

```
from sklearn.utils import class_weight
balanced_class_weights = class_weight.compute_class_weight('balanced',
                                                           np.unique(df_tr.labels),
                                                           df_tr.labels)
balanced_class_weights = {k:v for k, v in enumerate(balanced_class_weights)}
balanced_class_weights
```

```
{0: 0.7304347826086957, 1: 1.9090909090909092, 2: 0.9032258064516129}
```

We then train the neural networks as shown below:

```
df_tr, df_ts, std_scaler = fn_standardize_df(df_tr, df_ts)

model = multi_class_model_1
optimizer=keras.optimizers.SGD()

model_save_path =  data_path + 'NN_MODELS/MODEL_multiclass/'
batch_size, epochs = 32, 250

class_weight = balanced_class_weights

df_performance_model_multi = fn_NN_multi_clf(model, optimizer, df_tr, df_ts,
                                             model_save_path = model_save_path,
                                             batch_size=batch_size,
                                             epochs=epochs, class_weight = class_weight)

df_performance_model_multi.describe().round(3)
```

```
100% 250/250
```

	loss	acc	val_loss	val_acc	diff_loss	diff_acc
count	250.000	250.000	250.000	250.000	250.000	250.000
mean	0.268	0.927	0.223	0.932	0.067	0.020
std	0.275	0.096	0.229	0.071	0.058	0.038
min	0.036	0.460	0.086	0.449	0.000	0.000
25%	0.077	0.934	0.101	0.946	0.023	0.005
50%	0.139	0.964	0.115	0.957	0.045	0.011
75%	0.353	0.976	0.221	0.971	0.104	0.019
max	1.114	1.000	1.066	0.986	0.217	0.269

We then filter out the best two models as shown below:

```
dff = df_performance_model_multi

df1 = dff[dff.diff_acc < 0.03]
df2 = df1[df1.acc > 0.99]
df_filtered = df2[df2.loss < 0.04]

df_filtered
```

	loss	acc	val_loss	val_acc	diff_loss	diff_acc
multi_NN_1_epoch_217	0.037854	1.000000	0.102350	0.971014	0.064495	0.028986
multi_NN_1_epoch_218	0.036466	0.996032	0.105615	0.971014	0.069149	0.025017
multi_NN_1_epoch_242	0.037410	0.992063	0.106224	0.971014	0.068814	0.021049

MODEL TESTING:

We choose model "**multi_NN_1_epoch_217**" and check its generalization performance as shown below:

```
df_Xy_ = df_tr
model_ = keras.models.load_model(model_save_path + 'multi_NN_1_epoch_217.h5')

fn_test_model_multi_clf(df_Xy_, model_, label_encoder = labelencoder, NN = True)
```

```
==================
ACCURACY: 99.603
LOGLOSS: 0.02
==================
```

	precision	recall
class_Adelie	100.000	99.130
class_Chinstrap	97.778	100.000
class_Gentoo	100.000	100.000
MICRO	99.603	99.603
MACRO	99.259	99.710

```
df_Xy_ = df_ts
fn_test_model_multi_clf(df_Xy_, model_,
                        label_encoder = labelencoder,
                        NN = True)
```

```
==================
ACCURACY: 97.101
LOGLOSS: 0.099
==================
```

	precision	recall
class_Adelie	93.750	100.000
class_Chinstrap	100.000	92.857
class_Gentoo	100.000	96.000
MICRO	97.101	97.101
MACRO	97.917	96.286

As can be seen from the above results the model generalizes quite well. Note that we use the "**labelencoder**" keyword in the function above. This is the scikit learn label encoder object which was used to label encode the labels prior to training. Also, the "**NN**" key is set to True.

REGRESSION USING NEURAL NETWORKS:

We shall use the Boston Housing dataset to demonstrate how regression is performed using neural networks. We first configure the neural network using the code shown below. Note that it is almost the same configuration as the model used in the multiclass classification problem, except that:

- The output layer has only one neuron.
- The output layer has no activation function.

```
reg_model_1 = keras.models.Sequential(name = 'REG_NN_1')

reg_model_1.add(keras.layers.Dense(300, activation="relu", input_shape=(n_feats,)))
reg_model_1.add(keras.layers.Dropout(0.1))
reg_model_1.add(keras.layers.Dense(100, activation="relu"))
reg_model_1.add(keras.layers.Dropout(0.1))
reg_model_1.add(keras.layers.Dense(10, activation="relu"))
reg_model_1.add(keras.layers.Dense(1, activation=None))

reg_model_1.summary()
```

```
Model: "REG_NN_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_4 (Dense)              (None, 300)               2400
_________________________________________________________________
dropout_2 (Dropout)          (None, 300)               0
_________________________________________________________________
dense_5 (Dense)              (None, 100)               30100
_________________________________________________________________
dropout_3 (Dropout)          (None, 100)               0
_________________________________________________________________
dense_6 (Dense)              (None, 10)                1010
_________________________________________________________________
dense_7 (Dense)              (None, 1)                 11
=================================================================
Total params: 33,521
Trainable params: 33,521
Non-trainable params: 0
```

We use the function "**fn_NN_reg**" to train the model as shown in the image below. Note that the learning rate used is **1e-4** which is quite small compared to the earlier learning rates used. Smaller learning rates coupled with larger numbers of epochs is a good way to ensure that the gradient descent algorithm converges.

After training the model over 500 epochs, we filter out the best models using the same technique as in the earlier examples and inspect their performance using the CDF plots of the filtered models to choose the final model.

```
df_tr_standardized, df_ts_standardized, std_scaler = fn_standardize_df(df_tr, df_ts)

model = reg_model_1
lr = 1e-4
optimizer=keras.optimizers.SGD(lr)
model_save_path =  'drive/My Drive/1A_BOOK/CODE_6/MODELS/MODEL_REG/'
batch_size, epochs = 32, 500

df_performance_NN_reg = fn_NN_reg(model, optimizer,
                                  df_tr_standardized, df_ts_standardized,
                                  model_save_path,
                                  batch_size=batch_size, epochs=epochs)

df_performance_NN_reg.describe().round(3)
```

100% 500/500

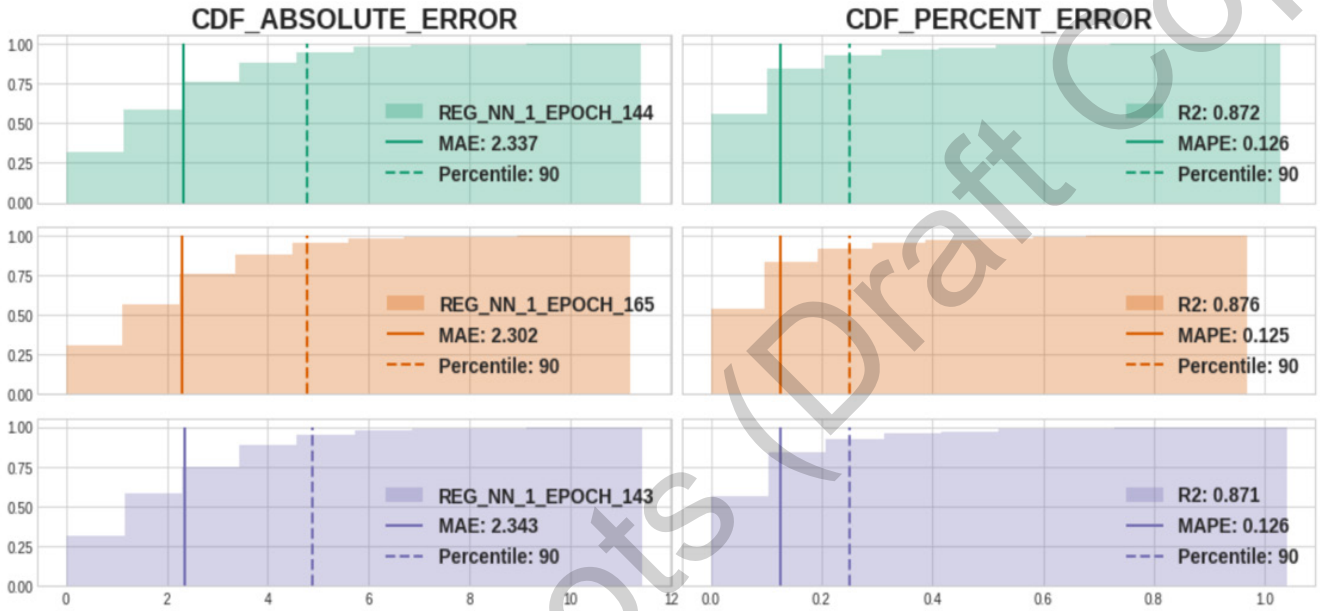
	loss	MAE	MAPE	val_loss	val_MAE	val_MAPE	diff_MAE	diff_MAPE
count	500.000	500.000	500.000	500.000	500.000	500.000	500.000	500.000
mean	20.586	2.998	15.517	25.718	3.187	16.513	0.244	1.296
std	61.708	2.426	10.766	60.542	2.345	10.103	0.156	0.774
min	7.762	2.129	11.408	15.306	2.615	13.802	0.001	0.001
25%	9.364	2.369	12.534	16.025	2.687	14.185	0.149	0.841
50%	10.426	2.516	13.245	16.728	2.755	14.573	0.249	1.336
75%	12.437	2.778	14.631	17.495	2.894	15.216	0.327	1.694
max	543.027	21.782	99.379	555.883	22.066	98.708	1.655	7.376

```
dff = df_performance_NN_reg

df1 = dff[dff.diff_MAPE < 0.5]
df2 = df1[df1.MAPE < 14.8]
df_filtered = df2[df2.MAE < 3].sort_values(by = "loss")[:3]

df_filtered
```

	loss	MAE	MAPE	val_loss	val_MAE	val_MAPE	diff_MAE	diff_MAPE
REG_NN_1_epoch_144	11.941509	2.765448	14.648210	17.363598	2.858465	15.063474	0.093017	0.415264
REG_NN_1_epoch_165	12.065818	2.782157	14.625677	17.319227	2.828204	14.922534	0.046047	0.296857
REG_NN_1_epoch_143	12.136522	2.764507	14.623817	17.380724	2.860204	15.076486	0.095697	0.452668

```
dict0_best_models = {
'REG_NN_1_epoch_144': keras.models.load_model(model_save_path + 'REG_NN_1_epoch_144.h5'),
'REG_NN_1_epoch_165': keras.models.load_model(model_save_path + 'REG_NN_1_epoch_165.h5'),
'REG_NN_1_epoch_143': keras.models.load_model(model_save_path + 'REG_NN_1_epoch_143.h5')}
```

```
df_tr_standardized, df_ts_standardized, std_scaler = fn_standardize_df(df_tr, df_ts)

fn_perform_models_data_reg(df_tr_standardized, dict0_best_models,
                           percentiles = [90], figsize = (14, 6))
```



Inspecting the CDF plots above, we choose **model_165** as it seems to give the best performance among the three models being compared. We check how well the model generalizes by testing it performance on the test set as shown below:

```
best_model = dict0_best_models['REG_NN_1_epoch_165']

fn_test_model_reg(df_ts_standardized, best_model)
```

	MAPE	R2	MAE
Performance:	0.151	0.751	2.834

As can be seen from above the model generalizes quite well and gives a better performance than the previous models used (Linear regression using Huber and RANSAC models).

6. NLP USING WORD2VEC EMBEDDINGS

NLP USING WORD2VEC EMBEDDINGS

WORD2VEC (W2V)

Word2vec is a text vectorization algorithm which has its basis in neural networks. It creates vector representations for individual words contained within a corpus (also called word embeddings), which is unlike the two techniques discussed previously which dealt with creating vector representations for documents within the corpus and used word count as the basis for vectorization.

There are two forms of implementations of the W2V algorithm:

1. Continuous Bag of Words (**CBOW**) technique
2. Skip Gram technique.

Both use a neural network architecture containing just one hidden layer, with no activation function. They basically differ in their input and output layer configurations and how the target vector is defined.

FOCUS WORDS AND CONTEXT WORDS:

Focus words and context words are at the core of how w2v algorithms generate word vectors. Given a particular language (say English) and all text associated with it, every word it contains has associations (or can be found to co-occur) with other words which share some semantic relationship. For instance, the word "**Happy**" may occur with other words like "**joy**", "**fun**", "**birthday**", "**wonderful**", "**new**", "**year**" and the like. The basic idea behind word2vec is this:

"The meaning of a word can be inferred by the company it keeps"

Focus word refers to any particular word being considered at the moment, that needs to be vectorized. Context words refer to the "company of other words" that this particular word being considered keeps.

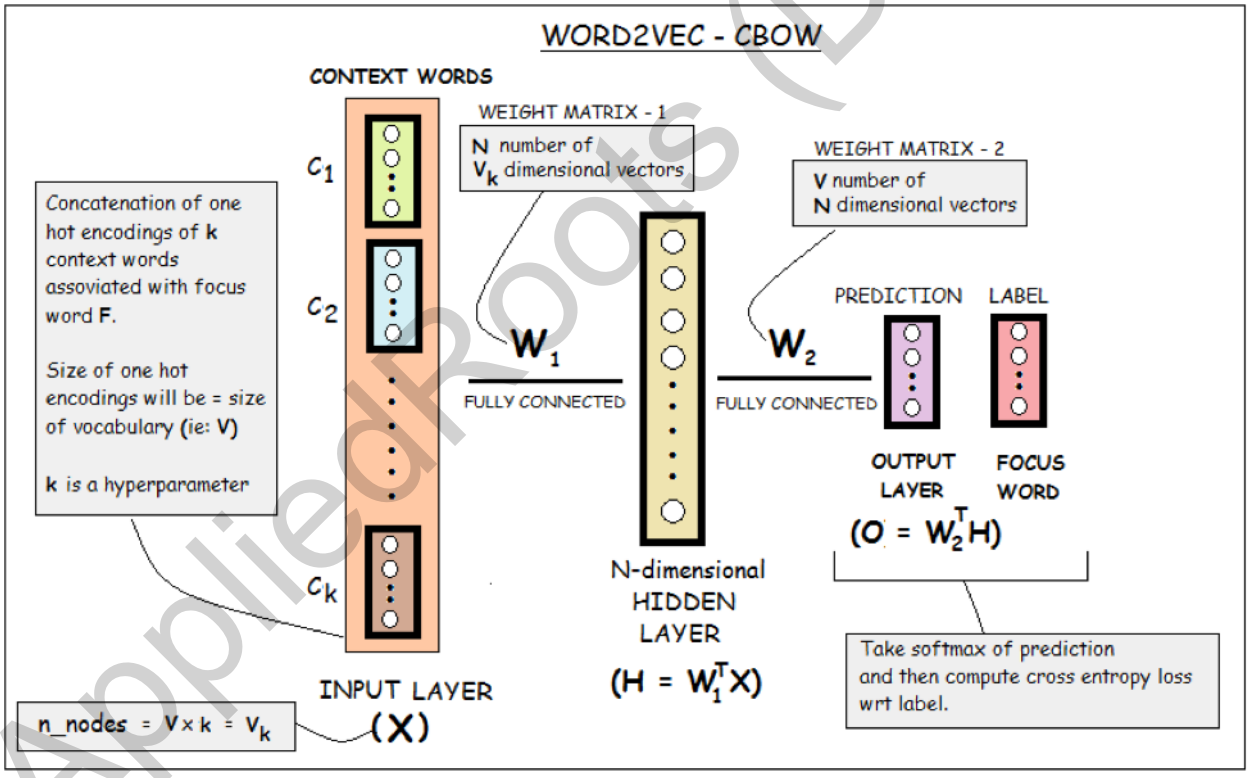
CBOW and Skip-gram techniques basically differ based on how they use Focus words and Context words. CBOW uses context words as inputs and focus words as their targets, whereas Skip-Gram does just the opposite - it uses Focus words as inputs and context words as their targets.

CONTINUOUS BAG OF WORDS (CBOW):

Continuous Bag of words aims at predicting the correct focus word given the corresponding set of context words. It involves the following steps:

1. All words in the corpus are one hot encoded (ie: Each word is represented as a binary sparse vector of length equal to the length of the vocabulary, and having only one cell in the entire vector having value equal to one).

2. We decide how many context words we want for every word in the corpus (**k**).
3. We create a dataset of Focus word and context words pairs.
4. We concatenate the one hot encodings of the context words of each focus word, to produce one huge context vector for each focus word.
5. We then train a basic neural network containing just one **N** dimensional hidden layer as shown in the image below. The input layer will have the same number of neurons as the dimensionality of the context vector and the output layer will have the same dimensionality as the one hot encoding of the focus word.
6. The training and evaluation of this neural network is conducted as per our discussions about neural networks in the previous chapters.
7. Step 6 results in two sets of weight matrices - one corresponding to the hidden layer (**W1**) and another correspondent to the output layer (**W2**).
8. The matrix **W2** will contain one **N** dimensional vector corresponding to each word in the vocabulary, thus we get a vector representation for each focus word.



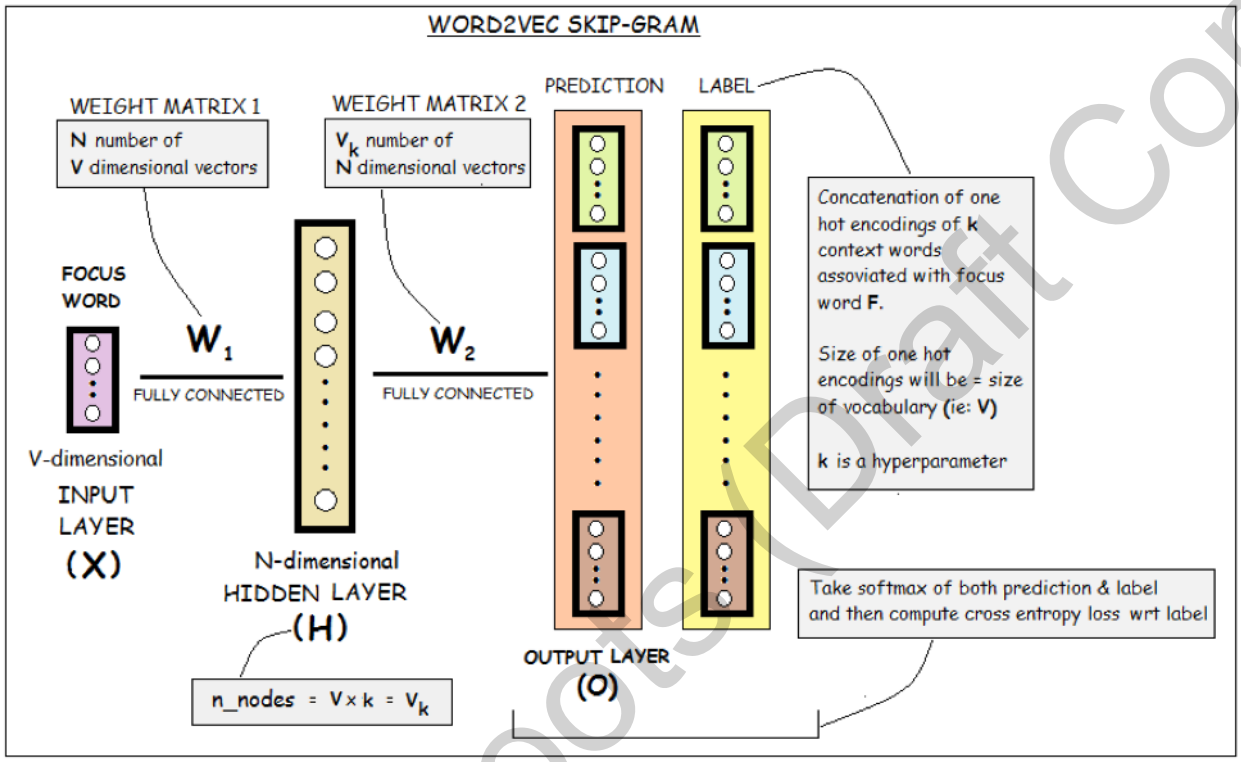
Note that:

1. The number of context words chosen (**k**) for each focus word and the number of neurons in the hidden layer (**N**) are hyperparameters that can be tuned to get the least loss between the predictions and the target.
2. The Hidden layer has no activation function and hence all non linearity is produced by the softmax in the output layer.

SKIP GRAM:

As discussed before, Skip gram is very similar to CBOW except it uses focus words as input vectors to predict context vectors, hence the steps 1 to 4 used for CBOW will apply for Skip gram also.

The neural network structure for Skip gram is shown in the image below:



Note that for Skip Gram:

1. The weight matrix **W1** will contain one **N** dimensional vector corresponding to each word in the vocabulary (instead of W2 as in CBOW) and thus we get a vector representation for each focus word.
2. Softmax is applied to the context vectors to get a probability distribution and only then is it used as a target.

The performance of CBOW and Skip gram models are quite similar, except that the Skip Gram model requires a larger train time, even though the number of trainable parameters for both models are the same. This is because in the Skip Gram technique, much larger softmax outputs have to be computed at the loss computation stage of every train iteration. Also, Skip Gram requires to be trained on larger corpuses to be effective in comparison, CBOW can produce quite effective vectors even with small corpuses.

CBOW is better at producing vectorized representation of common or frequently used words, but does not fare well while doing the same for rarely used words. Skip Gram does not have this problem. This is because CBOW uses multiple words (context words) to predict a single word (focus word) and hence if the target word is a rarely used word like “credulity”, the probability

of this word being the target word given its set of context words will be very low. In Skip Gram the input is the focus word itself and it is trained against its exact set of context words and hence this problem is reduced to some degree.

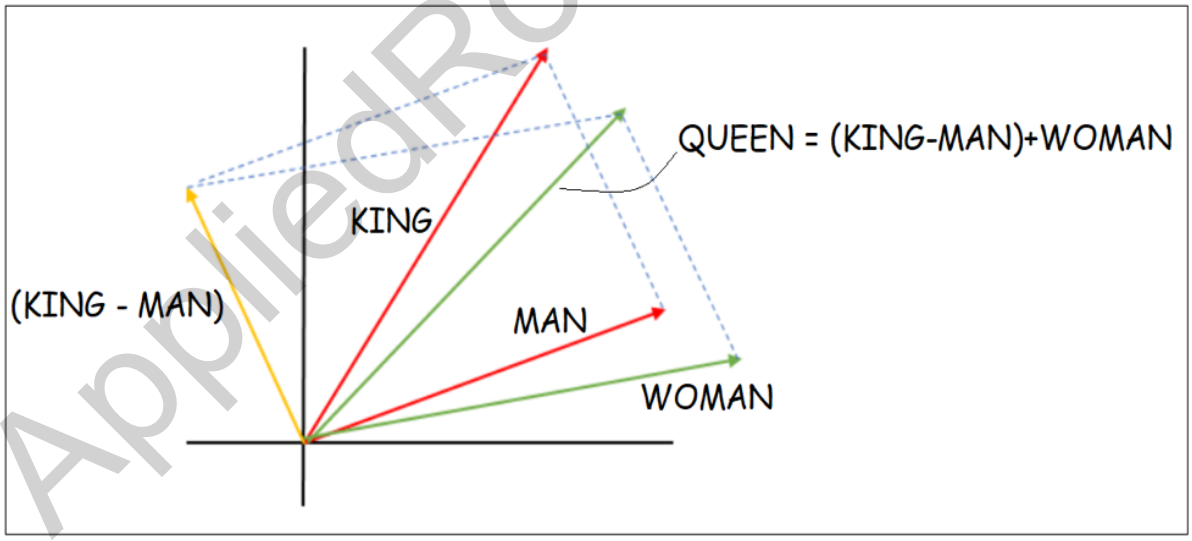
CAPTURING MEANING VIA SEMANTIC SIMILARITY OF WORDS:

The true power of w2v becomes more apparent when it is trained over corpuses that are so huge that they could be said to represent entire languages themselves. One such example is the pre-trained vectors made available by Google, which were obtained by training a w2v model on the **Google News** dataset containing about 100 billion words. The model contains 300-dimensional vectors for over 3 million words and phrases.

The word vectors obtained from the above model captures semantic relationships between words and thus seems to exhibit understanding of meaning. The following examples illustrate this:

- **vector('Paris') - vector('France') + vector('Italy')** results in a vector that is very close to vector('Rome')
- **vector('king') - vector('man') + vector('woman')** results in a vector that is very close to vector('queen')

In other other words, the w2v algorithm represents words in a complex high dimensional feature space that embeds semantic similarity and meaning of words. Thus linear algebraic operations between vectorized words are possible. The image below is a simplistic description of this:



Note that in the image above:

- The vectors representing Man and Woman are close
- The vectors representing King and Queen are close.
- The vector resulting from the vector operation King - Man + Woman results in a vector that is the same as (or very close to) the vector representing Queen.

CREATING DOCUMENT VECTORS FROM W2V WORD VECTORS:

As will become more clear later on in the chapter, most Natural Language Processing tasks require that text documents be vectorized just as was done using BOW or TF IDF methods. Since the W2V algorithm produces vectors for words, it is required that these word vectors are further processed to create document vectors. Two commonly used methods to do this are:

- Average W2V
- TF IDF weighted W2V

Average W2V as the name suggests, involves computing the W2V word vectors of all the words in the document and then representing the document using the vector average of all the word vectors thus computed.

TF IDF weighted W2V involves computing the W2V vectors and the individual TF IDF values of all the words in the document and then scaling each word vector using its corresponding TF IDF value. The individually scaled word vectors are then summed up to produce a resultant vector. This resultant vector is then scaled by reciprocal of the sum of all the individual TF IDF values of the words in the document.

TF IDF weighted W2V document vectors have been found to be quite effective representations of documents as shall be seen further on in this chapter.

THE W2V BASED DOCUMENT VECTORS:

For the sake of our sentiment classification task, we will be using **TF IDF weighted w2v** word vectors of the words in amazon reviews corpus, by using the freely available **google news w2v pretrained vectors**. We will be using the gensim library for loading the pretrained vectors as shown below. Once these are loaded, they can be used to obtain the w2v word vector corresponding to the words in the corpus.

In essence this vectorization object (w2v_google) is a compressed dictionary that maps over 3 million words and phrases to the corresponding 300-dimensional vectors learnt, by being trained on the **Google News dataset**.

```
from gensim.models import KeyedVectors

load_w2v = KeyedVectors.load_word2vec_format
name = 'GoogleNews-vectors-negative300.bin'
w2v_google = load_w2v(data_path + name, binary=True)

w2v_dims = w2v_google['wizard'].shape[0]
w2v_dims
```

```
300
```

TF IDF CODE:

As discussed previously in the chapter: "Intro to NLP", we create TF IDF matrix using the same corpus (amazon reviews) as shown below:

```
corpus = df_nlp_tr.reviews.values

kwargs = dict(min_df = 10, max_features = 3000,
              ngram_range = (1, 1), stop_words = 'english')
TFIDF_vectorizer = TfidfVectorizer(**kwargs).fit(corpus)
tr_tfidf_matrix = TFIDF_vectorizer.transform(corpus)

tr_tfidf_matrix.shape
```

```
(147659, 3000)
```

We then use the **TFIDF_vectorizer** derived from the trainset to create TF IDF vectors for the testset as shown below.

```
ts_tfidf_matrix = TFIDF_vectorizer.transform(df_nlp_ts.reviews.values)

ts_tfidf_matrix.shape
```

```
(36915, 3000)
```

We use the function **fn_w2v** shown below to create TF IDF weighted w2v word vectors. The **weight** argument of the function refers to the TF IDF value of the word currently being considered. The function returns the 300D word vector of any word fed to it. In case the word being considered does not exist in the vocabulary of the w2v model, the function returns a 300D zero vector.

```
def fn_w2v(word, weight):
    try:
        word_vec = w2v_model[word]
    except:
        word_vec = np.zeros(w2v_dims)
    return list(word_vec * weight)
```

We use the function **fn_tfidf_w2v** shown below to create TF IDF weighted document vectors. The function basically takes in the **tfidf_matrix** and **tfidf_features** (obtained using scikit learn's TfidfVectorizer) and the w2v_model loaded earlier as its arguments.

```python
def fn_tfidf_w2v(tfidf_matrix, tfidf_features, w2v_model, w2v_dims = 300):

    pbar = ProgressBar(max_value = tfidf_matrix.shape[0])
    list0_weighed_w2c_doc_vecs = []

    c, missed_rows = 0, []
    for tfidf_row in pbar(tfidf_matrix):

        non_zero_idxs = tfidf_row.toarray().squeeze().nonzero()[0]

        if len(non_zero_idxs) == 0:
            missed_rows.append(c)
            c += 1
            continue

        doc_weights = tfidf_row.toarray().squeeze()[non_zero_idxs]
        doc_words = np.array(tfidf_features)[non_zero_idxs]

        iter = zip(doc_words, doc_weights)
        weighed_word_vecs = np.array([fn_w2v(word, weight) for word, weight in iter])
        weighed_w2c_doc_vecs = sum(weighed_word_vecs)/(sum(doc_weights) + 1e-5)
        list0_weighed_w2c_doc_vecs.append(list(weighed_w2c_doc_vecs))

        pbar.update()
        c += 1

    arry0_weighed_w2c_doc_vecs = np.array(list0_weighed_w2c_doc_vecs)
    return arry0_weighed_w2c_doc_vecs, missed_rows
```

A

B

C

D

Avoid zero divide error

The function basically contains the following 4 main code blocks.

A. This code block uses the rows (ie: TF IDF document vectors) iterated from the TF IDF matrix and identifies the indexes in the current row that have non zero values. These indexes correspond to words present in that document.

B. Due to the fact that we have set **max_features** to be 3000 while creating TF IDF vectors, it could be possible that some documents have none of their words represented in the TF IDF vocabulary, thus resulting in code block A returning an empty list. Code block (B) skips all such documents and collects their indexes.

C. This code block extracts the words and their TF IDF values (ie: weights) that correspond to the indexes retrieved by code block A.

D. This code block simultaneously iterates over the **words** in the document and their corresponding **TF IDF** values and uses **fn_w2v** to compute the TF IDF weighted w2v document vectors for each row in the TFIDF matrix.

The code below shows the usage of **fn_tfidf_w2v** on the trainset's TF IDF matrix to arrive at the final form of our trainset. Now all the product reviews have been converted into 300D vectors which can be used for machine learning purposes. Note that three reviews corresponding to indexes 34482, 88576 and 143120 could not be processed due to lack of representation in the TF IDF vocabulary. We eventually delete the values corresponding to these indexes from the target variable.

```
tfidf_features = TFIDF_vectorizer.get_feature_names()
w2v_model = w2v_google

z = fn_tfidf_w2v(tr_tfidf_matrix, tfidf_features, w2v_model)
tr_arry0_tfidf_word_vecs, missed_rows = z

tr_arry0_tfidf_word_vecs.shape, len(missed_rows)
```

```
100% (147659 of 147659) |################| Elapsed Time: 0:06:41 Time:  0:06:41
((147656, 300), 3)
```

```
missed_rows
```

```
[34482, 88576, 143120]
```

We similarly obtain the final form of the test set as shown below:

```
tfidf_features = TFIDF_vectorizer.get_feature_names()
w2v_model = w2v_google

z = fn_tfidf_w2v(ts_tfidf_matrix, tfidf_features, w2v_model)
ts_arry0_tfidf_word_vecs, ts_missed_rows = z

ts_arry0_tfidf_word_vecs.shape, len(ts_missed_rows)
```

```
100% (36915 of 36915) |################| Elapsed Time: 0:01:44 Time:  0:01:44
((36915, 300), 0)
```

```
ts_missed_rows
```

```
[]
```

SENTIMENT CLASSIFICATION:

We configure the train and test sets as shown below. Note that we remove those values from the target variable that do not have corresponding TF IDF weighted w2v in X_train and X_test.

```
X_train = tr_arry0_tfidf_word_vecs
X_test = ts_arry0_tfidf_word_vecs

y_tr = df_nlp_tr.drop(missed_rows).ratings.values
y_ts = df_nlp_ts.drop(ts_missed_rows).ratings.values

X_train.shape, X_test.shape, y_tr.shape, y_ts.shape
```

```
((147656, 300), (36915, 300), (147656,), (36915,))
```

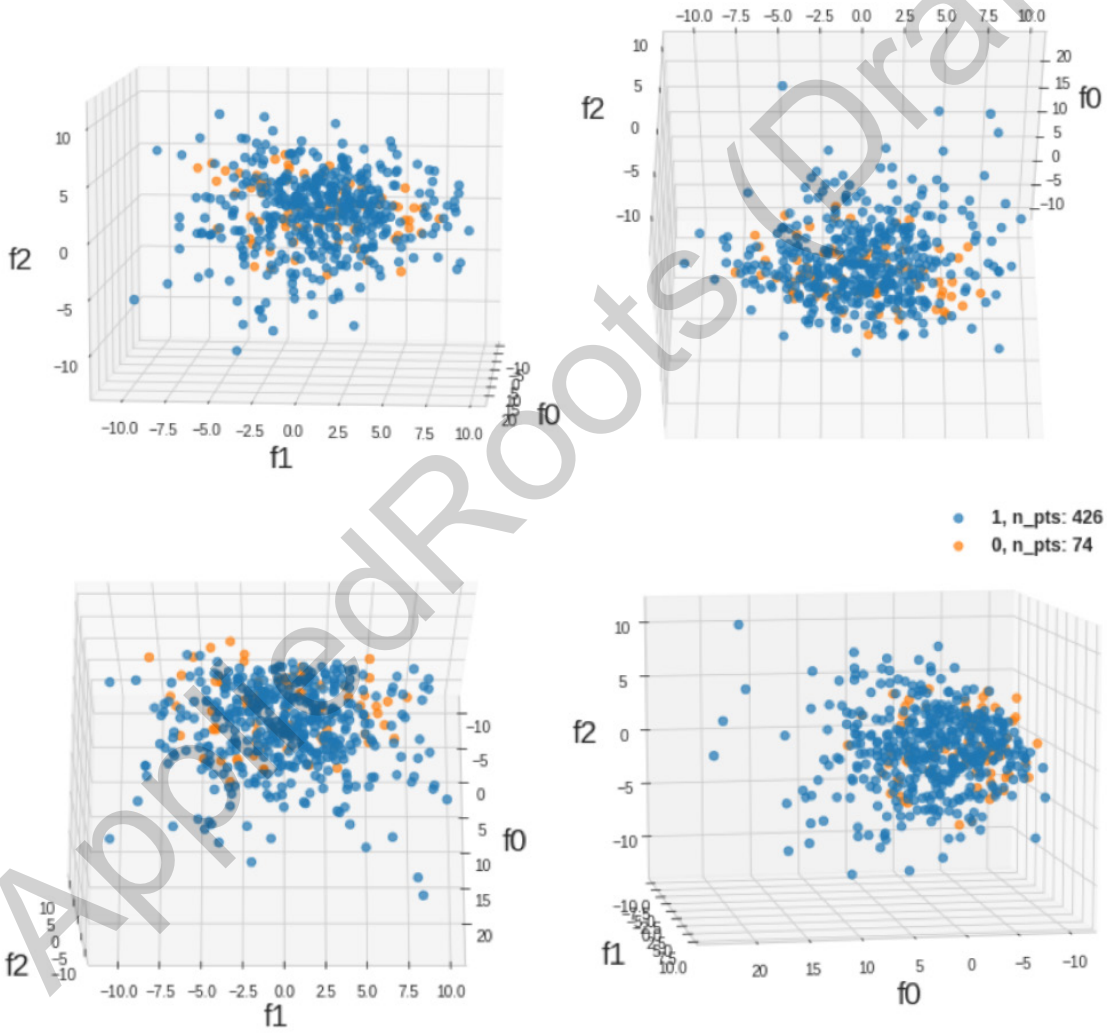
FINAL TRAIN AND TEST SETS:

```
df_tr = pd.DataFrame(X_train).assign(labels = y_tr)
df_ts = pd.DataFrame(X_test).assign(labels = y_ts)
```

VISUALIZING PCA DIMENSIONALLY REDUCED DATA:

We then visualize the data in two dimensions by dimensionally reducing it to 3D as shown below.

```
df_pca, pca_transformer = ai.fn_PCA(df_tr, reduce_dims = 100)

ai.fn_plot_3d_clf(df_pca.sample(500), alpha = 0.7)
```




```
df_pca.shape
```

```
(147656, 101)
```

As can be seen from the 3D scatter plots above, the data does not show any separation between the two classes in its dimensionally reduced form.

DIMENSIONALITY REDUCTION:

For the sake of computational ease, we dimensionally reduce the **df_tr** & **df_ts** to 100-dimensions as shown below.

```
X_ts_dim_reduced = pca_transformer.transform(df_ts.drop('labels', axis = 1).values)

df_tr_dimreduced = df_pca
df_ts_dimreduced = pd.DataFrame(X_ts_dim_reduced).assign(labels = df_ts.labels)
```

```
df_tr_dimreduced.shape, df_ts_dimreduced.shape
```

```
((147656, 101), (36915, 101))
```

K FOLD CROSS VALIDATION AND HYPERPARAMETER TUNING:

We perform grid search with K fold cross validation using the **Random Forest** classifier as shown in the code below.

```
from sklearn.ensemble import RandomForestClassifier

model_class = RandomForestClassifier
parameter_grid = dict(n_estimators = [300],
                      criterion = ['gini'],
                      max_depth = [None],
                  min_samples_split = [1000, 800, 600, 500, 400],
                      max_features = ['auto'],
                      class_weight = ['balanced'],
                      random_state = [0])

z = ai.fn_kfoldcv_clf_binary(df_tr_dimreduced, model_class, parameter_grid)

df_kfoldcv_gridsearch_2, dict0_model_instances_2, list0_invalid_models_2 = z
```

```
100% (5 of 5) |########################| Elapsed Time: 1:15:30 Time:  1:15:30
```

```
regd_columns = 'tr_prec_1 tr_rec_1 tr_prec_0 tr_rec_0 diff_rec_1 diff_rec_0'.split()
df_kfoldcv_gridsearch_2.loc[:, regd_columns].describe()
```

	tr_prec_1	tr_rec_1	tr_prec_0	tr_rec_0	diff_prec_1	diff_rec_1	diff_prec_0	diff_rec_0
count	60.000000	6.000000e+01	60.000000	60.000000	60.000000	6.000000e+01	60.000000	6.000000e+01
mean	0.952086	6.684102e-01	0.398009	0.728683	0.000473	5.711632e-04	0.001225	2.261845e-03
std	0.019981	2.093561e-01	0.128795	0.159166	0.000325	3.910447e-04	0.000823	1.571965e-03
min	0.906110	1.182396e-11	0.140834	0.333333	-0.000048	-1.182396e-11	-0.000006	-5.528962e-04
25%	0.953056	6.666667e-01	0.363997	0.726815	0.000005	-3.941292e-12	0.000006	-6.010387e-12
50%	0.957638	7.733642e-01	0.364012	0.791296	0.000714	8.040296e-04	0.001829	3.438317e-03
75%	0.957643	7.733750e-01	0.364683	0.791320	0.000720	8.276775e-04	0.001852	3.486397e-03
max	1.000000	7.842560e-01	0.713613	1.000000	0.000769	1.580470e-03	0.001871	3.558528e-03

We then filter the best models as shown below:

```
dff = df_kfoldcv_gridsearch_2
df1 = dff[dff.diff_rec_0 < 0.08]
df2 = df1[df1.tr_rec_0 > 0.6]
df_filtered = df2[df2.tr_rec_1 > 0.6]
df_filtered.loc[:, regd_columns]
```

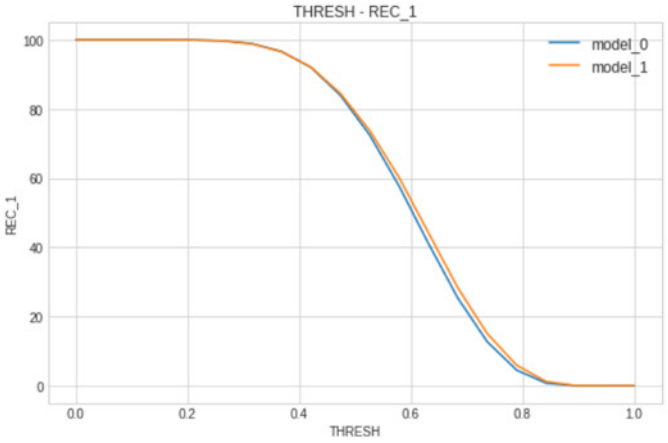
	tr_prec_1	tr_rec_1	tr_prec_0	tr_rec_0	diff_rec_1	diff_rec_0
model_0	0.953315	0.787121	0.370654	0.764847	0.007757	0.061265
model_1	0.955172	0.797641	0.384639	0.771628	0.009195	0.076990

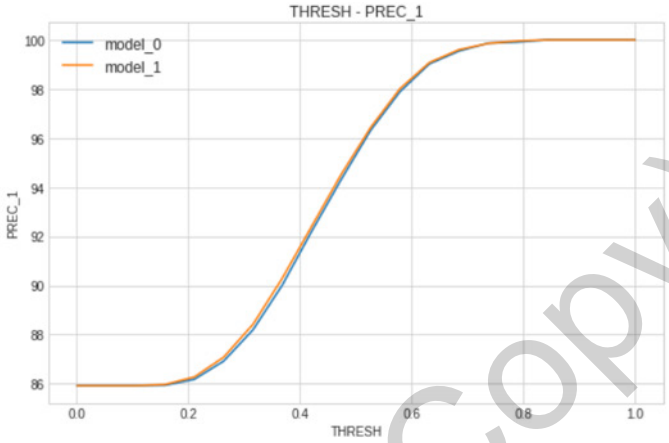
CHECK MODEL PERFORMANCE USING PRECISION & RECALL CURVES:

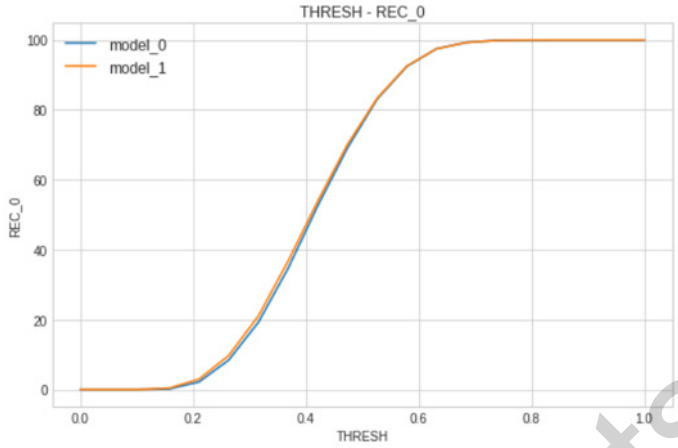
We inspect the precision and recall curves for the top models as shown below.

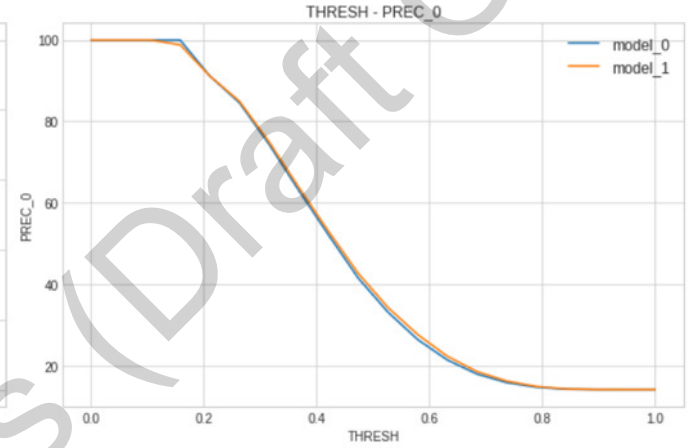
```
df_tr_standard, df_ts_standard, scaler = ai.fn_standardize_df(df_tr_dimreduced, df_ts_dimreduced)

list0_model_names = list(df_filtered.index)
X_train, y_train = df_tr_standard.iloc[:, :-1].values, df_tr_standard.iloc[:, -1].values
%time list0_models = [dict0_model_instances_2[i].fit(X_train, y_train) for i in list0_model_names]

list0_thresholds = np.linspace(0, 1, 20)
legend = list0_model_names

%time ai.fn_performance_models_data(list0_models, df_tr_standard, list0_thresholds, legend)
```

```
CPU times: user 13min 23s, sys: 436 ms, total: 13min 24s
Wall time: 13min 22s
CPU times: user 20min 18s, sys: 3.95 s, total: 20min 22s
Wall time: 20min 19s
```





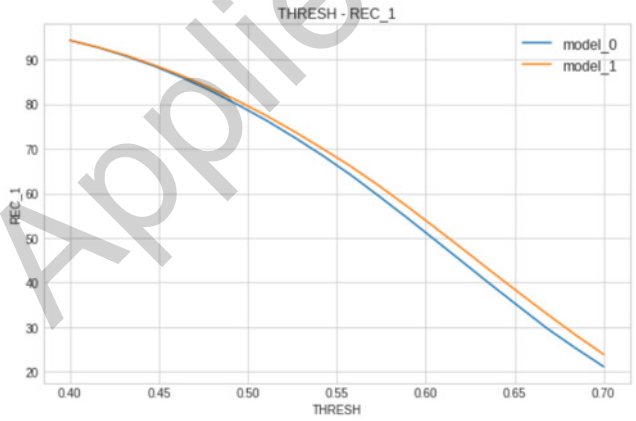


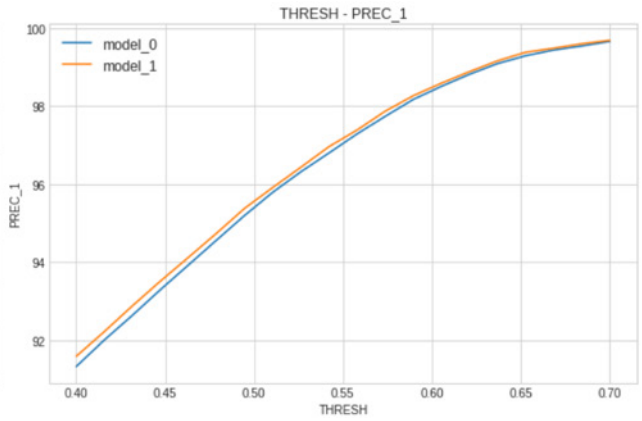


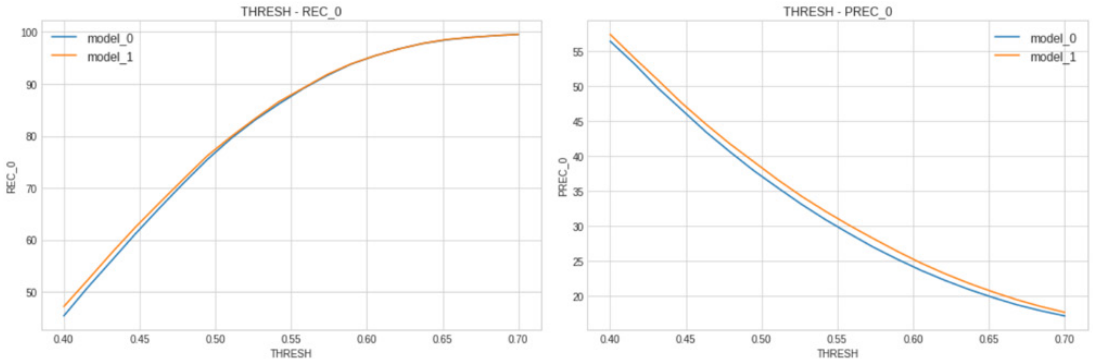
We zoom into the threshold range of 0.4 to 0.7 to get a better perspective:

```
1   list0_thresholds = np.linspace(0.4, 0.7, 20)
2   legend = list0_model_names
3
4   %time ai.fn_performance_models_data(list0_models, df_tr_standard, list0_thresholds, legend)
```

```
CPU times: user 19min 55s, sys: 3.37 s, total: 19min 59s
Wall time: 19min 56s
```







Assuming that we are optimizing for the recall of the negative reviews (class 0), we inspect the plots shown above and choose model_1 and threshold at a value of 0.525.

We then check how well the chosen model generalizes as shown below:

```
df_Xy_ = df_tr_standard
model_ = dict0_model_instances_2['model_1'].fit(X_train, y_train)
thresh = 0.525

ai.fn_test_model_binary_clf(df_Xy_, model_, threshold_class_1 = thresh)
```

```
--------------------
LOGLOSS : 0.5282
ACCURACY: 75.367
--------------------
```

	prec	rec
class_0	34.458	83.034
class_1	96.383	74.110

```
df_Xy_ = df_ts_standard

ai.fn_test_model_binary_clf(df_Xy_, model_, threshold_class_1 = thresh)
```

```
--------------------
LOGLOSS : 0.541
ACCURACY: 73.434
--------------------
```

	prec	rec
class_0	31.538	75.707
class_1	94.831	73.061

As can be seen from the above outputs the model generalized as expected (diff in recall between train and test sets, ie: diff_rec_0 is around 0.07, which is the same as the condition chosen during filtering off the best models). All other metrics generalize well across the train and test sets.

The negative recall of the model is around 76% on the test set, with a corresponding positive recall of around 73%.

7.CLUSTERING -1 (KMEANS++)

KMEANS++

In the chapter fundamentals of Machine Learning - 1, we discussed **Lloyd's algorithm** which is used in K Means clustering and the concept of Within Cluster Sum of Squares **(WCSS)** or **inertia**.

Given a particular cluster, the WCSS is defined as the sum of squared distances of all the points within that cluster to its centroid.

$$\underset{\text{(For cluster S)}}{WCSS} = \sum_{x \in S} \| x - \mu \|^2$$

Where μ is the centroid of the points in cluster **S** .

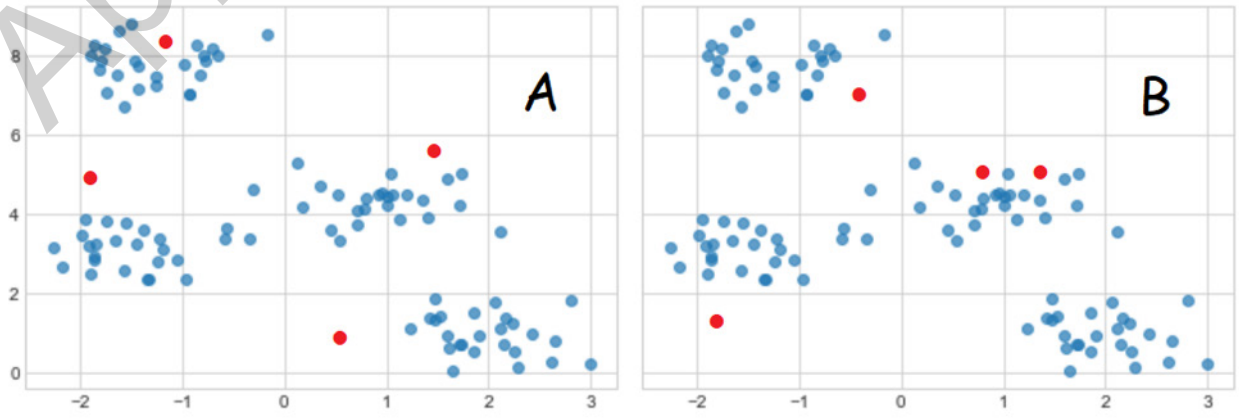
Essentially, WCSS measures the variability of the data points within each cluster. In general, a cluster that has a small sum of squares is more compact than a cluster that has a large sum of squares.

Lloyd's algorithm is the heuristic algorithm used to minimize the WCSS of a dataset. It consists of the following steps:

1. Initialize **K** centroids by randomly selecting **K** unique points from the feature space.
2. Assign a centroid for every data point in the dataset, based on least distance. Thus K initial clusters are created with reference to these randomly initiated centroids.
3. Recompute centroids for every cluster thus obtained.
4. Repeat steps 2 & 3 until convergence (ie: the Total WCSS value does not change much).

PROBLEMS DUE TO RANDOM INITIALIZATION:

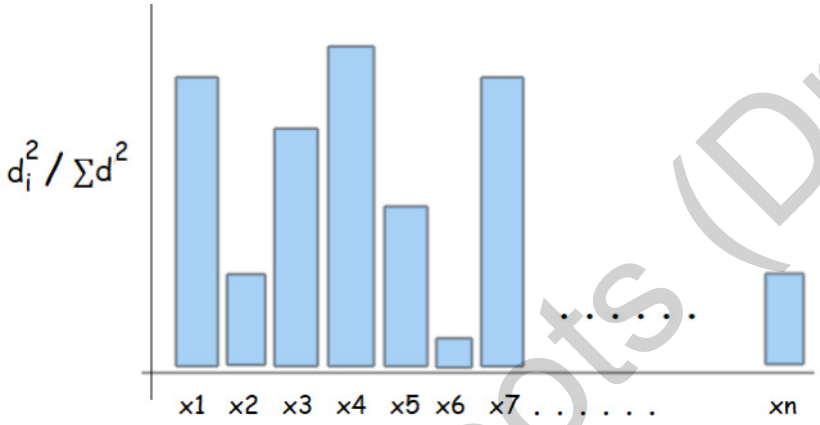
The effectiveness of Llyod's algorithm mainly depends on the first K randomly selected points. If the randomly chosen points are such that they are located as shown in image A below, the Llyod's algorithm will generally provide good clustering. But if the initially chosen points are located as shown in image B (Some centroids too close to each other), then the algorithm might result in suboptimal clustering.



K MEANS ++ INITIALIZATION TECHNIQUE:

The K Means ++ algorithm is an initialization technique for Lloyd's algorithm. It aims at overcoming the random initialization problem just discussed, by choosing K initial centroids such that they are appropriately far apart from each other. It involves the following steps:

1. Randomly choose the first centroid.
2. Choose the next centroid using the following procedure:
   a. For each data point compute the distance(s) with respect to all the previously chosen centroids and choose the smallest distance to represent it ($\mathbf{d_i}$ = min(distances from centroids) - in case of the first step there will be only one centroid).
   b. Create a probability distribution where each data point's probability is proportional to the **normalized square** of distance **d** associated with it, as shown below.



3. Randomly choose a point from the above distribution and use it as the next centroid.
4. Repeat steps 2 and 3 till K centroids are got
5. Use these centroids as the initial centroids in **Llyod's** algorithm.

Using squared distances increases the separation between points far away from current centroid and points close to it and thus using normalized squared distance as probability, increases the chance of randomly selecting a "faraway" point.

We will be using the **scikit learn** library to perform K means clustering on the **breast cancer data set** shown below.

```
import numpy as np
import pandas as pd

data = load_breast_cancer()
df_cancer = pd.DataFrame(data['data'])
df_cancer.columns = ['f' + str(i) for i in range(len(df_cancer.columns))]
df_cancer = df_cancer.assign(labels = data['target'])

display(df_cancer.head())
print(df_cancer.shape)
```

	f0	f1	f2	f3	f4	...	f25	f26	f27	f28	f29	labels
0	17.99	10.38	122.80	1001.0	0.11840	...	0.6656	0.7119	0.2654	0.4601	0.11890	0
1	20.57	17.77	132.90	1326.0	0.08474	...	0.1866	0.2416	0.1860	0.2750	0.08902	0
2	19.69	21.25	130.00	1203.0	0.10960	...	0.4245	0.4504	0.2430	0.3613	0.08758	0
3	11.42	20.38	77.58	386.1	0.14250	...	0.8663	0.6869	0.2575	0.6638	0.17300	0
4	20.29	14.34	135.10	1297.0	0.10030	...	0.2050	0.4000	0.1625	0.2364	0.07678	0

5 rows × 31 columns

```
(569, 31)
```

The dataset contains thirty features/dimensions representing different biological measurements and has labels indicating whether the observations (rows) correspond to a positive case of cancer or not (1 & 0).

The scikit learn library provides two classes for K means clustering:

1. The **KMeans** class
2. The **MiniBatchKMeans** class

THE KMEANS CLASS:

The KMeans class is scikit learn's main tool for K Means Clustering. Once an instance of this class is instantiated, by providing the required hyperparameter values, it can be fit to the data we want to cluster by using the fit method (ie: The fit method applies the K means algorithm on the data). Note that the data needs to be standardized before fitting the KMeans instance on to it.

```
from sklearn.preprocessing import StandardScaler

df_X = df_cancer.iloc[:, :-1]
std_X = StandardScaler().fit_transform(df_X.values)
df_X_std = pd.DataFrame(std_X)
```

The "trained"/fitted KMeans instance is now capable of predicting the cluster of any other new data point of the same kind. We use the **predict** method for doing this. To know the cluster that each point in the original data belongs to, we use the predict method on it, this returns an array containing cluster labels corresponding to each data point. This is demonstrated in the code shown below.

We use the **cluster_centers_** attribute of the fitted KMeans instance to access the cluster centers of the clusters detected by the K Means clustering. This returns an array of cluster centers indexed as per their labels.(ie: **kmeans_model.cluster_centers_[0]** will give centroid of cluster 0 and so on).

```
from sklearn.cluster import KMeans

kmeans = KMeans(n_clusters = 3, init = 'k-means++', n_init = 5)
kmeans_model = kmeans.fit(std_X)
cluster_per_pt = kmeans_model.predict(std_X)

display(cluster_per_pt[: 20])
display(kmeans_model.cluster_centers_.shape)
```

```
array([2, 2, 2, 0, 2, 0, 2, 0, 0, 0, 1, 0, 2, 1, 0, 0, 1, 0, 2, 1])
(3, 30)
```

In the code shown above, we can see three parameters that were defined while instantiating the KMeans instance named "**kmeans**". These three parameters/kwargs are the most relevant specifications of the instance. The K Means algorithm requires that we specify the number of clusters expected within the data and it clusters the data based on this information. The **n_cluster** kwarg lets us specify this. It is set to three in the above example.

The init and n_init parameters specify the random selection procedure to be used for selecting the first set of centroids. The init parameter specifies the method to use to initialize the first set of centroids. We have two choices for this parameter: "**k-means++**" and "**random**". Scikit learn by default specifies this as the former. The **n_init** parameter is used to specify how many times the initiation method specified by the **init** parameter has to be repeated. In the example above this is specified as five. This means that the K Means algorithm will be run five times for five different kmeans++ initiations and the initiation that yielded the least WCSS/**inertia** value will be used. Scikit learn specifies a default value of ten for this parameter.

As can be seen from the output of the code above, the **predict** method returns an array of cluster labels (0, 1, 2) corresponding to each data point and the **cluster_centers_** attributes gives us three thirty dimensional cluster centers.

THE MINIBATCHKMEANS CLASS:

In the example above we randomly chose **n_cluster** to be three. In reality one needs to use the "**elbow method**" to determine the optimal number of clusters. This is where the **MiniBatchKmeans** class comes in. The MiniBatchKMeans is a variant of the KMeans algorithm which uses a slightly modified version of Lloyd's algorithm, where instead of using the entire dataset at each iteration, to update the centroids, it uses smaller random samples of fixed size. The mini-batch k-means produces results that are generally quite comparable to that of the standard K-Means algorithm, but not as good. **MiniBatchKmeans** saves a lot of compute time and makes it possible to cluster large sized datasets without taking up much memory.

We choose a range of **n_clusters** values and create a MiniBatchKmeans instance for each of these n_clusters values. We then fit each of these instances to the data and compute the resulting WCSS/inertia values. The **n_clusters** value corresponding to the **elbow** of the **n_clusters Vs inertia** plot gives us the optimal n_clusters.

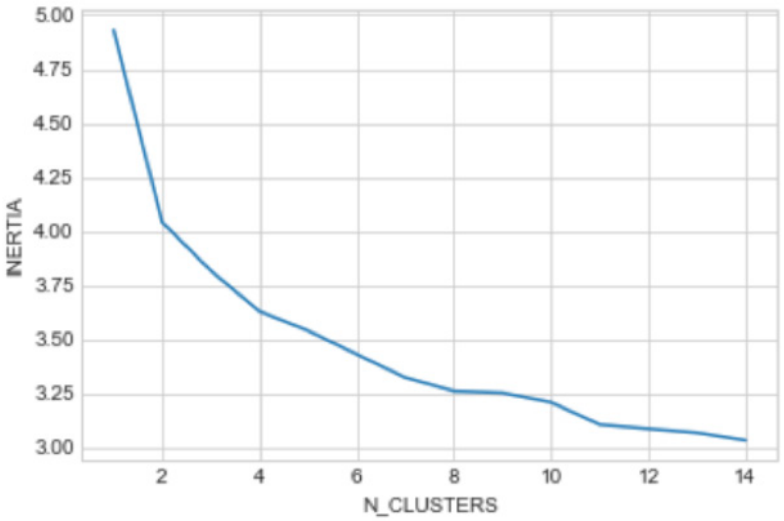
The two functions shown below implement the elbow method described below. The function **fn_inertia** computes the inertia resulting from choosing a particular value for n_clusters. The **fn_elbow_plot** uses **fn_inertia** to compute the inertia values for a range of n_clusters values and plots the "**elbow**" curve described earlier.

```
def fn_inertia(data_points, n_clusters, batch_size):

    from sklearn.metrics.pairwise import paired_distances

    # MINI BATCH K_MEANS CLUSTERING:
    kmeans = MiniBatchKMeans(n_clusters = n_clusters,
                             batch_size = batch_size)
    kmeans = kmeans.fit(data_points)

    # Array containing cluster centers corresponding to each data point:
    cluster_centers = [kmeans.cluster_centers_[idx] for idx in kmeans.labels_]
    cluster_centers = np.array(cluster_centers)

    #---[dist_of_each_point and its cluster].mean():
    inertia = paired_distances(data_points, cluster_centers, metric='euclidean')

    return inertia.mean()
```

```
def fn_elbow_plot(data_points, n_clusters_choices, batch_size):

    results = []
    pbar = ProgressBar()
    for n_clusters in pbar(n_clusters_choices):

        avg_var = fn_avg_var(data_points, n_clusters, batch_size)
        results.append([n_clusters, avg_var])

    df_sorted_kmeans = pd.DataFrame(results, columns = 'K loss'.split())
    df_sorted_kmeans = df_sorted_kmeans.sort_values(by=['loss'])
    sns.lineplot(df_sorted_kmeans.K.values, df_sorted_kmeans.loss.values)
    plt.xlabel('K')
    plt.ylabel('LOSS')
    plt.show()
```

```
data_points = df_X_std.values
n_clusters_choices = range(1, 15)
batch_size = 500

fn_elbow_plot(data_points, n_clusters_choices, batch_size)
```

```
100% (14 of 14) |########################| Elapsed Time: 0:00:00 Time:  0:00:00
```



The "elbow" of the plot shown above is at n_clusters = 2 and hence we will use this value to finally cluster the data points using the function **fn_kmeans_model** shown below.

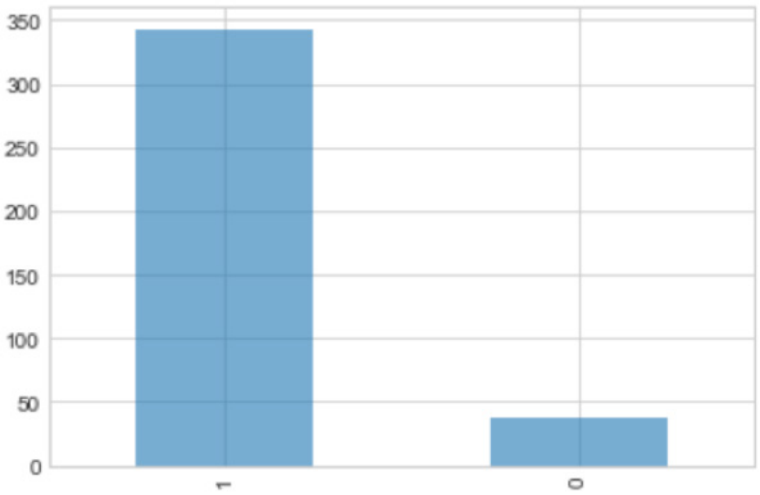
```
def fn_kmeans_model(data_points, n_clusters):

    kmeans = KMeans(n_clusters = n_clusters)
    kmeans_model = kmeans.fit(data_points)

    n_points = [1 for i in range(len(data_points))] #--For using groupby in dataframe
    clusters = kmeans_model.labels_

    df = pd.DataFrame().assign(clusters = clusters, n_points = n_points)
    df0_clusters_info = df.groupby('clusters').sum()

    display(df0_clusters_info)
    return  kmeans_model
```

The function shown above performs KMeans clustering on the given data points using the specified n_clusters and returns the k-means model/instance that was fitted to the data. This can be used for prediction purposes. Also, the function displays the clusters and the number of clusters information. This is demonstrated in the code shown below.

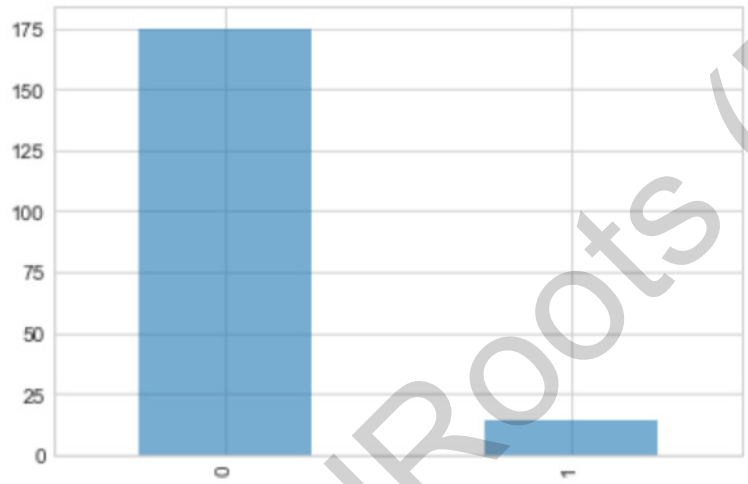
```
n_clusters = 2
kmeans_model = fn_kmeans_model(data_points, n_clusters)
```

clusters	n_points
0	189
1	380

```
df_compare_clusters = df_cancer.assign(clusters = kmeans_model.labels_).iloc[:, -2:]
df = df_compare_clusters

df[df.clusters == 0].labels.value_counts().plot.bar(alpha = 0.6)
plt.show()
```




```
df[df.clusters == 1].labels.value_counts().plot.bar(alpha = 0.6)
plt.show()
```



As can be inferred from the plots shown above,there is a slight amount of non cancerous points clustered as cancerous and vice versa, but the clustering in general seems to reflect the two groups contained within the data.

ADVANTAGES AND LIMITATIONS OF K MEANS:

In general k-means is quite an effective clustering method despite its simplicity. It runs relatively quickly and scales quite effectively to medium sized datasets. But it has its own limitations.

Since k-means relies totally on the centroids to define its clusters, It can only handle relatively simple spherical shapes (it assumes distances along all directions are equally important) and can be quite sensitive to small dimensional data having odd shapes. This will be discussed further in the chapter named **Dbscan**. This limitation of k means reduces as the dimensionality of the data increases, because distances start becoming nonlinear.

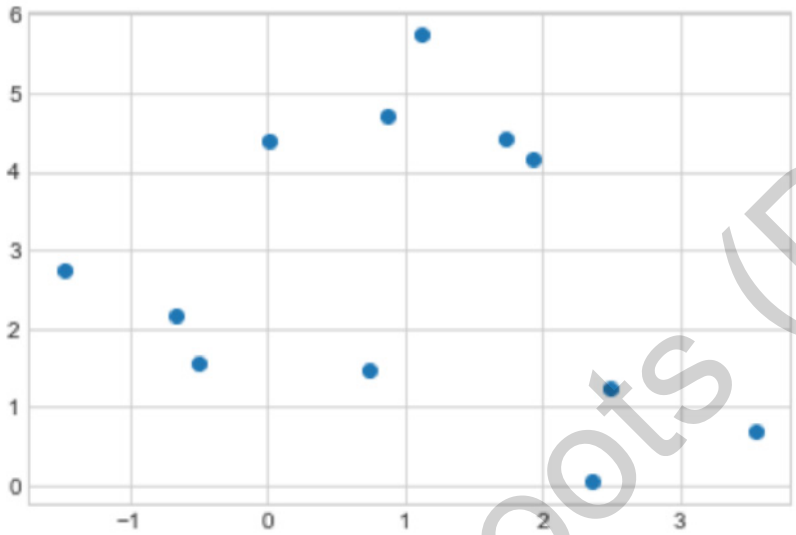
The k-means algorithm can be prone to the effect of **outliers**, during the selection of the initial set of centroids. We use the technique of sampling from a distribution proportional to the **normalized squared** distance of the data from the centroid, because doing so reduces the chances of choosing an outlier (instead of directly choosing the point with the largest probability).

8. CLUSTERING -2 (HIERARCHICAL CLUSTERING)

HIERARCHICAL CLUSTERING

Hierarchical clustering as the name suggests involves creating a hierarchy of clusters. It can be explained using the toy data shown below. The data has twelve points that form three clusters.

```
from sklearn.datasets import make_blobs

X, y = make_blobs(random_state=0, n_samples=12)
plt.scatter(X[:, 0], X[:, 1])
plt.show()
```

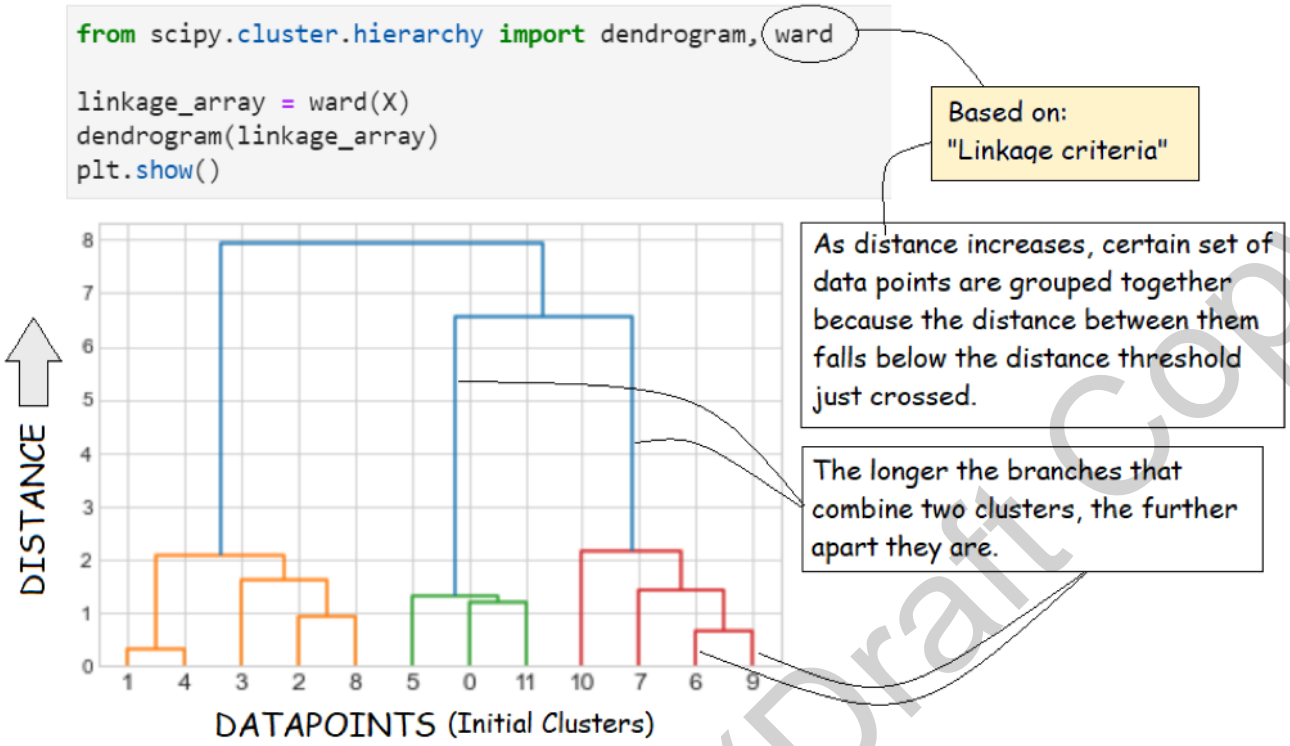


Hierarchical clustering can be applied to the above data using the following steps:

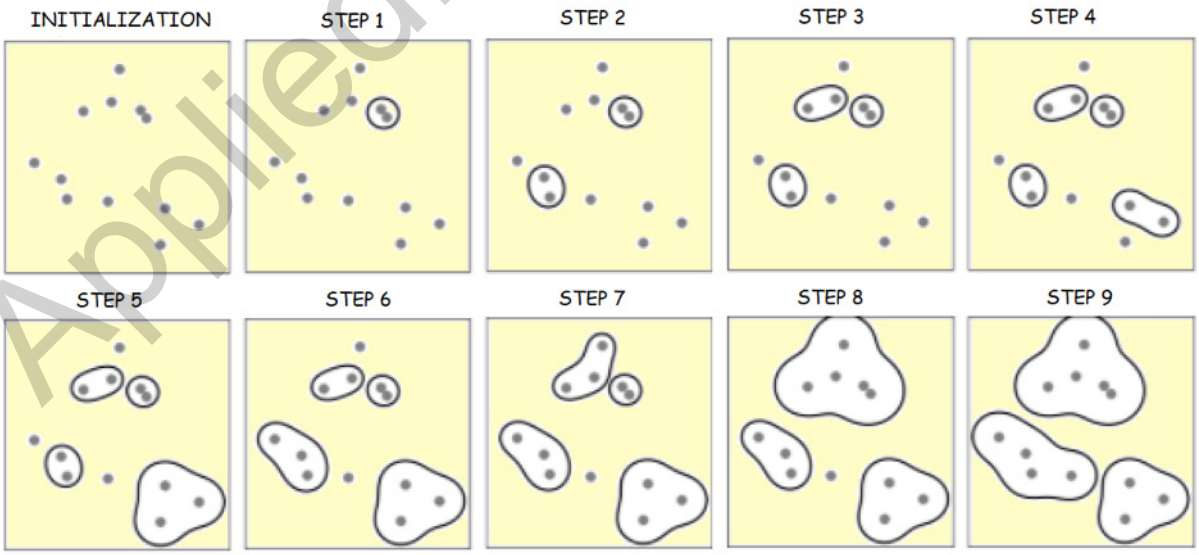
1. We start by considering each data point as a cluster by themselves.
2. Then those clusters that are closest/most similar to each other are grouped together forming bigger clusters. The closeness/similarity between clusters is based on a criteria called '**Linkage criteria**". This will be discussed further on in the chapter.
3. Step 2 is repeated till we ultimately end up with a single cluster that encompassess all the data. This process can be visualized using the **scipy** library's **dendrogram** visualization as shown below.

```
from scipy.cluster.hierarchy import dendrogram, ward

linkage_array = ward(X)
dendrogram(linkage_array)
plt.show()
```

```
from scipy.cluster.hierarchy import dendrogram, ward

linkage_array = ward(X)
dendrogram(linkage_array)
plt.show()
```



As can be seen from the dentogram shown above, at zero distance (ie: At max similarity) there are 12 clusters (ie: each point is considered as a cluster). As distance is increased and it passes certain thresholds, the number of clusters decreases and the size of the resulting set of clusters increases. So at distance = 1, we have around 4 small clusters, at distance just above 2, we have three larger clusters, at distance = 7 we have 2 clusters and at distance = 8, we have one large cluster. The image below visualizes the progression of forming three clusters in the example above as a sequence of nine steps (Distance zero to just above two). This bottom up Hierarchical clustering technique is called **Agglomerative** clustering.



One can also use a top-down approach. Where we start by considering the entire dataset as a single cluster and this cluster is split using a suitable clustering algorithm like kmeans. This procedure is applied recursively to each of the resulting clusters until each data point is in its own cluster. This form of hierarchical clustering is known as **Divisive** clustering. Divisive clustering is seldom used, as the splitting conditions for splitting larger clusters into smaller ones is not as clear as it is in the case of Agglomerative clustering, instead one has to use some other subroutine/algorithm to split the clusters recursively.

LINKAGE CRITERIA:

The linkage criterion refers to the merge strategy used to merge similar clusters. The algorithm will merge the pairs of clusters that minimize the value specified by this criterion. The following linkage criteria are used for merging clusters:

Ward: minimizes the sum of squared differences within all clusters. It is a variance-minimizing approach and in this sense is similar to the k-means objective function.

Complete: minimizes the maximum distance between observations of pairs of clusters.

Single linkage: minimizes the distance between the closest observations of pairs of clusters.

Average: minimizes the average of the distances between all observations of pairs of clusters.

DATA:

We use the same data used in the k-means example (ie: Breast Cancer data) as shown below.

```
import numpy as np
import pandas as pd

data = load_breast_cancer()
df_cancer = pd.DataFrame(data['data'])
df_cancer.columns = ['f' + str(i) for i in range(len(df_cancer.columns))]
df_cancer = df_cancer.assign(labels = data['target'])

display(df_cancer.head())
print(df_cancer.shape)
```

	f0	f1	f2	f3	f4	...	f25	f26	f27	f28	f29	labels
0	17.99	10.38	122.80	1001.0	0.11840	...	0.6656	0.7119	0.2654	0.4601	0.11890	0
1	20.57	17.77	132.90	1326.0	0.08474	...	0.1866	0.2416	0.1860	0.2750	0.08902	0
2	19.69	21.25	130.00	1203.0	0.10960	...	0.4245	0.4504	0.2430	0.3613	0.08758	0
3	11.42	20.38	77.58	386.1	0.14250	...	0.8663	0.6869	0.2575	0.6638	0.17300	0
4	20.29	14.34	135.10	1297.0	0.10030	...	0.2050	0.4000	0.1625	0.2364	0.07678	0

5 rows × 31 columns

```
(569, 31)
```

We standardize the data as shown below.

```
dff_X = df_cancer.iloc[:, :-1]
std_X = StandardScaler().fit_transform(dff_X.values)
df_X_std = pd.DataFrame(std_X)
```

Scikit learn allows us to perform Agglomerative clustering in one of the following two ways.

1. By explicitly specifying the number of clusters (ie: the **n_clusters** parameter/kwarg) that exist in the data and allowing the algorithm to continue the clustering process until the specified number of clusters is reached.
2. By specifying the threshold for the linkage criteria (ie: the **distance_threshold** parameter/kwarg), beyond which the clustering algorithm stops. In this method, the number of clusters is determined by the number of clusters formed until the specified threshold is reached.

Shown below is an example of Agglomerative clustering using the first method described above. In the code shown, three of the most relevant **parameters** for creating an instance of the **AgglomerativeClustering** class are demonstrated. We randomly specify **n_clusters** = 5.

```
from sklearn.cluster import AgglomerativeClustering

agg = AgglomerativeClustering(n_clusters = 5,
                      distance_threshold = None,
                       compute_distances = True)
agg_model = agg.fit(std_X)
```

Note that the **distance_threshold** parameter is set to **None** in the code above. The **compute_distances** parameter when set to True, computes the distances between the clusters during the agglomeration process.

The model fitted clustering model **agg_model** has three basic attributes: **n_clusters_, distances_ and labels_**, as demonstrated below.

```
agg_model.n_clusters_
```

```
5
```

```
d = agg_model.distances_
d.max(), d.min(), d.mean()
```

```
(102.01433991004352, 1.0061149471356947, 4.587269467170271)
```

```
agg_model.labels_[:10]
```

```
array([2, 4, 2, 0, 4, 0, 4, 0, 0, 0], dtype=int64)
```

Shown below is an example of Agglomerative clustering using the second method described earlier. Note that the **distance_threshold** parameter is now set to twenty (randomly specified for demonstration purposes) and the **n_clusters** parameter is set to None.

```
agg2 = AgglomerativeClustering(n_clusters = None,
                     distance_threshold = 20,
                      compute_distances = True)
agg_model_2 = agg2.fit(std_X)
```

```
agg_model_2.labels_[:10]
```

```
array([2, 4, 2, 0, 4, 0, 4, 0, 0, 0], dtype=int64)
```

FINDING THE OPTIMAL NUMBER OF CLUSTERS:

In the two agglomerative clustering examples described earlier, we chose the **n_clusters / distance_threshold** parameters randomly just for the sake of demonstration. We can determine the optimal values for **n_clusters** using the steps outlined below:

1. Given data to be clustered, we fit an instance of the **AgglomerativeClustering** class to it, with the n_clusters parameter specified to one and the **compute_distances** parameter set to True.

```
agg = AgglomerativeClustering(n_clusters = 1,
                     distance_threshold = None,
                      compute_distances = True)
agg_model = agg.fit(std_X)
```

2. We then use the **distances_** attribute to find the max, min and mean distances encountered during the clustering process.

```
d = agg_model.distances_
d.max(), d.min(), d.mean()
```

```
(102.01433991004352, 1.0061149471356947, 4.587269467170271)
```

3. For a range of **distance_threshold** parameter values between the mean and max distance values computed above:
   a. Fit agglomerative clustering models to the data (or a suitably sized sample of the data) at each of these threshold values.
   a. Determine the resulting **n_clusters** corresponding to each of these **distance_threshold** values.

We use the function **fn_n_clusters_thresh** shown below to perform the steps a and b described above. This function takes in a list of **distance_threshold** values and returns a DataFrame that shows the number of clusters corresponding to each of these thresholds.

```
def fn_n_clusters_thresh(list0_thresholds):

    rows = []
    for thresh in list0_thresholds:

        agg = AgglomerativeClustering(n_clusters = None,
                                      distance_threshold = thresh)
        agg_model = agg.fit(std_X)
        n_clusters = agg_model.n_clusters_
        rows.append([thresh, n_clusters])

    df = pd.DataFrame(rows, columns = ['thresh', 'n_clusters'])
    return df
```

```
list0_thresholds = np.linspace(4.5, 130, 20)

df_thresh_cluster = fn_n_clusters_thresh(list0_thresholds)
display(df_thresh_cluster.iloc[:10], df_thresh_cluster.iloc[10:])
```

	thresh	n_clusters
0	4.500000	157
1	11.105263	34
2	17.710526	17
3	24.315789	10
4	30.921053	6
5	37.526316	3
6	44.131579	3
7	50.736842	3
8	57.342105	2
9	63.947368	2

	thresh	n_clusters
10	70.552632	2
11	77.157895	2
12	83.763158	2
13	90.368421	2
14	96.973684	2
15	103.578947	1
16	110.184211	1
17	116.789474	1
18	123.394737	1
19	130.000000	1

4. Since the range of **distance_threshold** values specified are equally spaced, the number of times a "**n_clusters**" value repeats itself, is an indicator of the distance of the linkages described in the **dentogram**. From the DataFrames shown above, we can see n_clusters = 2 repeats the most and hence the optimal number of clusters is two. This can be further summarized using the function **fn_agg_summarize** shown below.

```
def fn_agg_summarize(df_thresh_cluster):

    s_n_repeats = df.n_clusters.value_counts()
    n_clusters = s_n_repeats.index
    n_repeats = s_n_repeats.values

    df_agg1 = pd.DataFrame().assign(n_clusters = n_clusters,
                                    n_repeats = n_repeats)
    df_agg2 = df.groupby('n_clusters').min()
    df_summary = df_agg1.merge(df_agg2, on = 'n_clusters')

    return df_summary
```

```
df_summary = fn_agg_summarize(df_thresh_cluster)
df_summary
```

	n_clusters	n_repeats	thresh
0	2	7	57.342105
1	1	5	103.578947
2	3	3	37.526316
3	157	1	4.500000
4	17	1	17.710526
5	10	1	24.315789
6	6	1	30.921053
7	34	1	11.105263

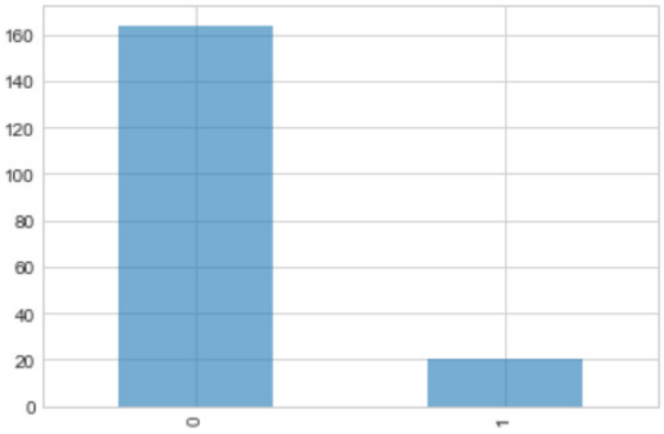
CHECKING CLUSTERING PERFORMANCE :

Based on the information of the above summary, we choose **n_clusters** = 2 and fit an agglomerative clustering model to the data as shown below.

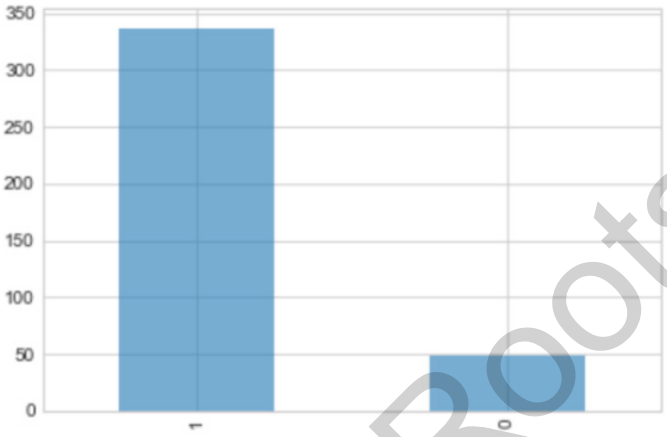
```
agg = AgglomerativeClustering(n_clusters = 2)
agg_model = agg.fit(std_X)
```

We then check the clustering performance of the model with respect to the actual labels of the data set as shown below.

```
cluster_per_pt = agg_model.labels_
df_compare_label_clusters = df_cancer.assign(clusters = cluster_per_pt).iloc[:, -2:]
df = df_compare_label_clusters

df[df.clusters == 0].labels.value_counts().plot.bar(alpha = 0.6)
plt.show()
```




```
df = df_compare_label_clusters

df[df.clusters == 1].labels.value_counts().plot.bar(alpha = 0.6)
plt.show()
```



As can be inferred from the plots shown above,there is a slight amount of non cancerous points clustered as cancerous and vice versa, but the clustering in general seems to reflect the two groups contained within the data.

ADVANTAGES AND LIMITATIONS OF AGGLOMERATIVE CLUSTERING:

Though Agglomerative clustering is relatively easy to understand and implement, it is not scalable to large data sets, because the clustering time required increases exponentially as data size increases. This is because at each step it has to consider all the possible merges.

The agglomerative clustering method can only be used to cluster the data it was fitted on and cannot be used to predict on new data, because it has no cluster reference points (ex: like the centroids in the case of K-means).

9. CLUSTERING – 3 (DBSCAN)

DBSCAN

DBSCAN stands for Density Based Spatial Clustering of Applications with Noise. The main idea used within this clustering algorithm is that clusters are regions that are densely populated separated from other clusters by regions that are sparsely populated within the data space.

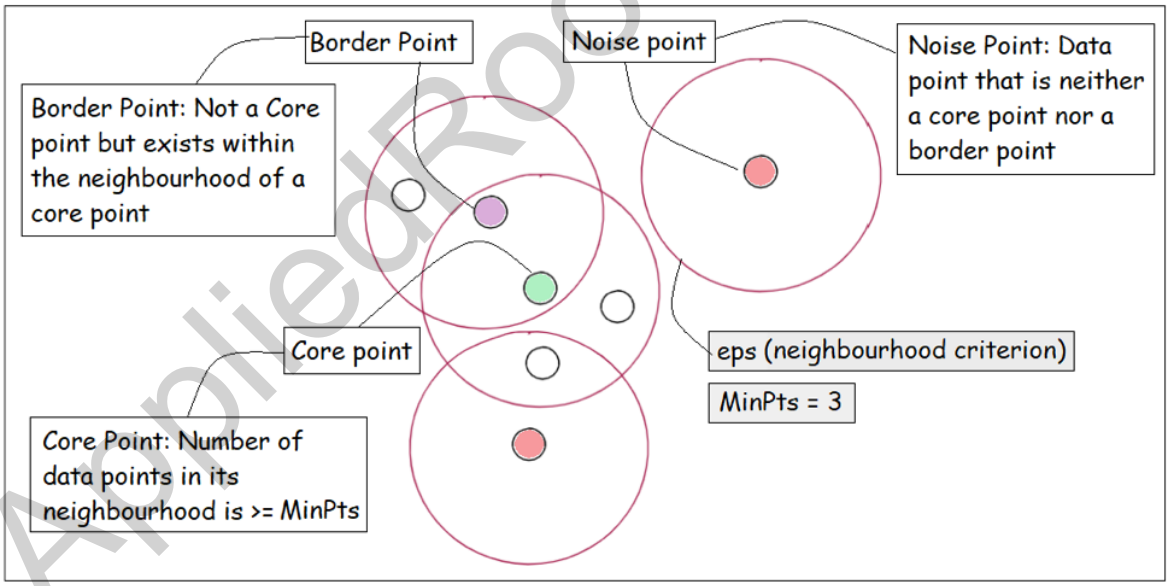
A data point is considered to be existing within a **dense** region, if the number of data points in its neighbourhood is greater than or equal to some predefined minimum threshold, where :

1. The neighbourhood of a point is defined by the size of a hyper-sphere at that point. The size of the hyper-sphere is defined by its radius epsilon or **eps**.
2. The threshold for the minimum number of points is called minimum points or **MInPts**

So, for a given **eps** value, if the number of data points in its neighbourhood is greater than or equal to **MInPts**, the data point is considered to be existing within a **dense** region in the data space, else it is considered to be existing within a **sparse** region.

CORE, BORDER AND NOISE POINTS:

The DBSCAN algorithm works by first classifying data points into core, boundary and noise points, based on the density of the region they exist in. The figure shown below describes the basis of this classification.



Points that possess dense neighbourhoods are classified as core points. Points that possess sparse neighbourhoods, but exist within the neighbourhood of at least one core point are called border points. These are data points that exist at the border of clusters. Points that possess sparse neighbourhoods and that do not exist in the neighbourhood of core points are called noise points. They do not form a part of any cluster and represent noise as the name suggests.

DENSITY CONNECTED POINTS:

When the distance between two core points is less than the eps value specified, the line that connects them is called a **density edge**. Given a core point, all other core points that are sequentially connected to it via density edges form a cluster belonging to that core point.

THE DBSCAN ALGORITHM:

The DBSCAN algorithm consists of the following steps:

1. For the specified MinPts and eps values, classify each data point as core, border or noise point
2. Remove all noise points from the data.
3. Randomly choose a core point and assign all other core points that are density connected to it as belonging to this randomly chosen core point's cluster.
4. Repeat step 3 on the remaining core points that are not a part of any cluster, till all core points are exhausted
5. Assign every border point to the nearest cluster.

CHOOSING THE RIGHT PARAMETER VALUES:

The clustering results of the dbscan algorithm are primarily dependent upon the values assigned to parameters **eps** and **MinPts**.

The eps value determines what it means for points to be neighbours (ie: The range of the neighbourhood). For a given MinPts, the smaller the eps, the lesser the number of core points formed during clustering and the more the number of noise points. Too small and eps may result in all points being classified as noise.

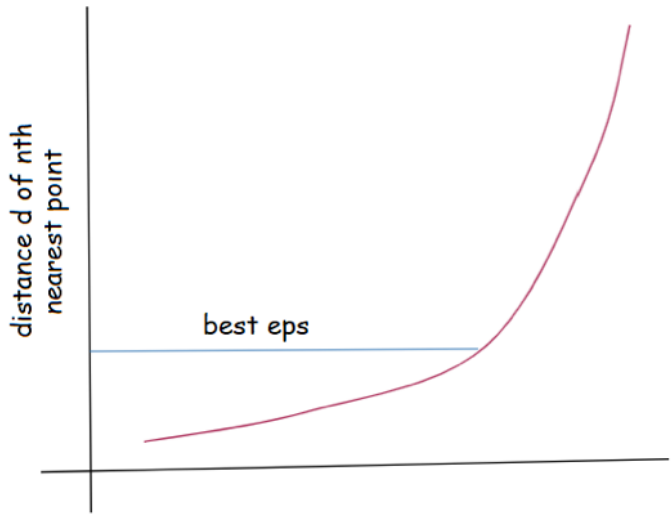
The MinPts value determines whether a given neighborhood is dense or not . For a given eps value, the higher the MinPts, the lesser the number of dense neighborhoods that are formed.

Different data, require different eps and MinPts values depending on its density variations and size. Usually the MinPts value is specified first and then the optimal eps value is determined using a process that is kind of similar to the one used in KMeans to find the best k. Choosing the right MinPts values requires domain knowledge. But the following rule of thumb can sometimes be helpful for choosing the right MinPts value:

1. MinPts should be >= d, where d is the number of dimensions/features in the data.
2. For large datasets, typically MinPts should be >= 2d.
3. Use larger MinPts value if we know that the data is noisy.

The steps outlined below describe the process for finding the best eps for a **given** MinPts value.

1. For every data point **x** determine the distance **d** of the **n**th nearest point from **x**. Where n = MinPts.
2. Sort in ascending order all the distances (d) computed for the various points in the dataset.
3. Plot a curve of sorted distances and choose the distance at the elbow of the curve to be the eps value for clustering the data.



DATA:

We use the same data used in the previous examples (ie: Breast Cancer data) as shown below.

```
import numpy as np
import pandas as pd

data = load_breast_cancer()
df_cancer = pd.DataFrame(data['data'])
df_cancer.columns = ['f' + str(i) for i in range(len(df_cancer.columns))]
df_cancer = df_cancer.assign(labels = data['target'])

display(df_cancer.head())
print(df_cancer.shape)
```

	f0	f1	f2	f3	f4	...	f25	f26	f27	f28	f29	labels
0	17.99	10.38	122.80	1001.0	0.11840	...	0.6656	0.7119	0.2654	0.4601	0.11890	0
1	20.57	17.77	132.90	1326.0	0.08474	...	0.1866	0.2416	0.1860	0.2750	0.08902	0
2	19.69	21.25	130.00	1203.0	0.10960	...	0.4245	0.4504	0.2430	0.3613	0.08758	0
3	11.42	20.38	77.58	386.1	0.14250	...	0.8663	0.6869	0.2575	0.6638	0.17300	0
4	20.29	14.34	135.10	1297.0	0.10030	...	0.2050	0.4000	0.1625	0.2364	0.07678	0

5 rows × 31 columns

```
(569, 31)
```

We standardize the data as shown below.

```
dff_X = df_cancer.iloc[:, :-1]
std_X = StandardScaler().fit_transform(dff_X.values)
df_X_std = pd.DataFrame(std_X)
```

We use the function **fn_best_eps** shown below to implement the steps outlined earlier for finding the best eps for a given MinPts value. The **sample_size** argument of the function is used to specify the size of the sample that would be selected randomly from the data for computing the best eps. This is handy incase of large datasets.

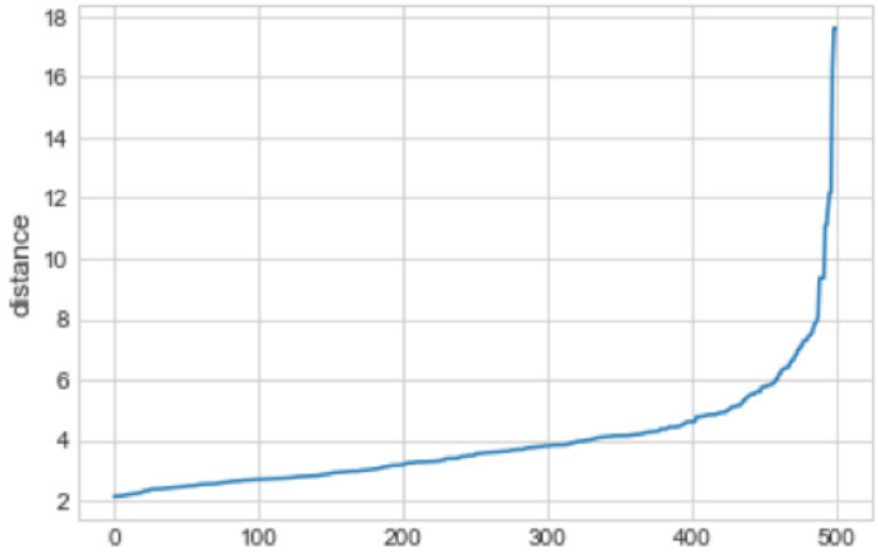
```
def fn_best_eps(data, min_samples, sample_size, metric = 'euclidean'):

    from progressbar import ProgressBar
    from sklearn.metrics.pairwise import euclidean_distances, cosine_distances

    pbar = ProgressBar()
    distances = []
    for i in pbar(range(sample_size)):

        point = data.sample(1).values
        if metric == 'euclidean':
            dists = euclidean_distances(data, point.reshape(1, -1))
        if metric == 'cosine':
            dists = cosine_distances(data, point.reshape(1, -1))
        dist_min_samples = sorted(dists)[min_samples]
        distances.append(dist_min_samples)

    x, y = range(len(distances)), sorted(distances)
    plt.ylabel('distance', fontsize = 12)
    plt.plot(x, y)
```

Note that the DBSCAN algorithm provides for many choices of distance metric. In the function shown above, two most relevant distance metrics can be used: **Euclidean** distance and Cosine distance.

We choose MinPts = 35 based on the thumb rule described earlier for choosing MinPts values. For this MinPts value we create an eps plot for euclidean distance as shown below:

```
data = df_X_std
min_samples, n_iters = 35, 500

fn_best_eps(data, min_samples, n_iters, metric = 'euclidean')
```

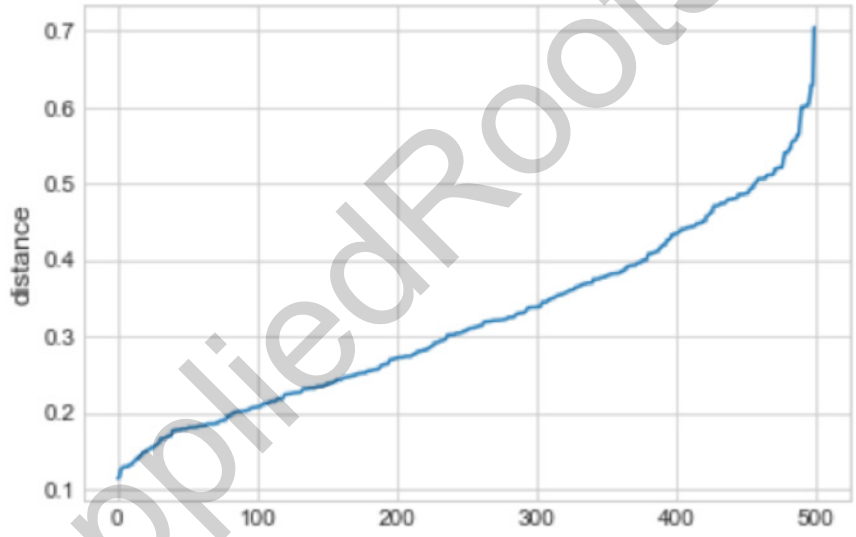
```
100% (500 of 500) |######################| Elapsed Time: 0:00:01
```



We repeat the step described above using Cosine distance as shown below.

```
data = df_X_std
min_samples, n_iters = 35, 500

fn_best_eps(data, min_samples, n_iters, metric = 'cosine')
```

```
100% (500 of 500) |######################| Elapsed Time: 0:00:01
```



From the above plots we see that the optimal range of distance values is in the range of 2 - 18 for euclidean distance and between 0.1 to 0.7 for cosine distance. We use this information in the **function fn_summarize_dbscan** shown below to determine the best combination of eps and MinPts.

```
def fn_summarize_dbscan(df_X_std, param_grid_MinPts_eps):

    X_std = df_X_std.values
    list0_rows = []

    pbar = ProgressBar()
    list0_param_combos= fn_combinations([*param_grid_MinPts_eps.values()])
    for MinPts, eps, metric in pbar(list0_param_combos):

        model = DBSCAN(eps=eps, min_samples = MinPts, metric = metric).fit(X_std)
        dff = df_X_std.assign(clusters = model.labels_)
        sr = dff.clusters.value_counts()
        n_noise = sr.values[0]
        row = [eps, MinPts, metric, n_noise, len(sr.values[1:])]
        list0_rows.append(row)

    df_dbscan_results = pd.DataFrame(list0_rows)
    df_dbscan_results.columns = 'eps MinPts metric n_noise n_clusters'.split()

    return df_dbscan_results.sort_values(by = 'n_noise')
```

This function shown above takes in a range of eps and MinPts values and fits instances scikit learn's DBSCAN class to the data using all possible combinations of eps and MinPts and summarizes the clustering results in a DataFrame that shows the number of noise points and number of clusters corresponding to each eps - MinPts combination. The code shown below demonstrates this.

```
param_grid_MinPts_eps = dict(MinPts = list(range(3, 50)),
                             eps = np.linspace(0.1, 1, 20),
                          metric = ['euclidean', 'cosine'])


df_dbscan_trials = fn_summarize_dbscan(df_X_std, param_grid_MinPts_eps)
df_dbscan_trials.head()
```

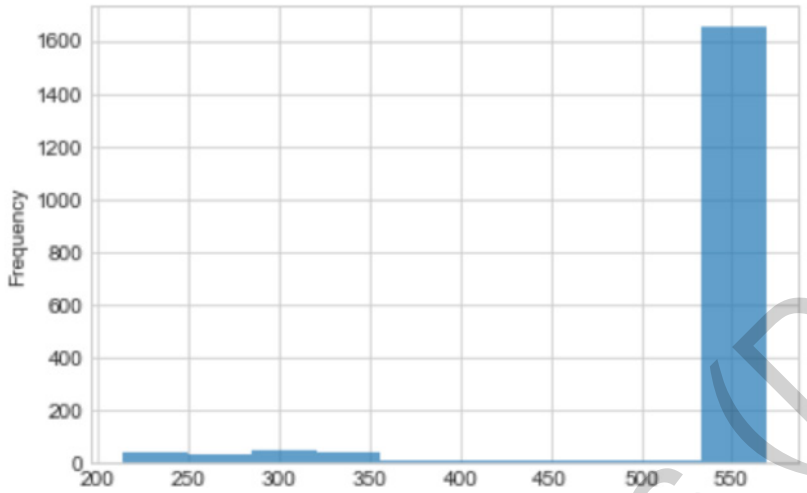
```
100% (1880 of 1880) |####################| Elapsed Time: 0:00:17 Time:  0:00:17
```

	eps	MinPts	metric	n_noise	n_clusters
83	0.147368	5	cosine	214	4
845	0.194737	24	cosine	221	2
805	0.194737	23	cosine	221	2
43	0.147368	4	cosine	222	5
123	0.147368	6	cosine	223	3

From the output shown above we see that DBSCAN classifies a major part of the data as noise. The combination of eps = 0.147 and MinPts = 5 (1st row) produces the least number of noise points and it results in four clusters.

We then check the distribution of noise among the clusters produced by each of the DBSCAN models used in the code above as shown below.

```
df_dbscan_trials.n_noise.plot.hist(alpha = 0.7)
plt.show()
```



Using the information from the plot above, we filter the results based on best criteria as demonstrated in the code shown below.

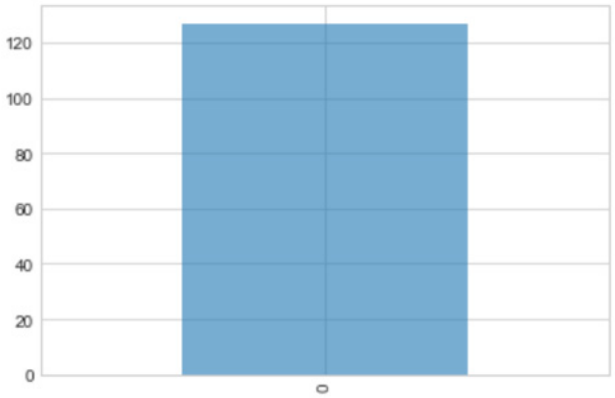
```
condn_1 = df_dbscan_trials.n_clusters >= 2
condn_2 =  df_dbscan_trials.n_noise <= 225

df_dbscan_trials[condn_1 & condn_2].head(3)
```

	eps	MinPts	metric	n_noise	n_clusters
83	0.147368	5	cosine	214	4
845	0.194737	24	cosine	221	2
805	0.194737	23	cosine	221	2

The three models shown in the output above produce the least noise. All of them use cosine distance. We choose the model in the second row of the DataFrame (eps = 0.195 and MinPts = 24) for our prediction purposes. We then check the clustering performance of this chosen model as demonstrated in the code below.

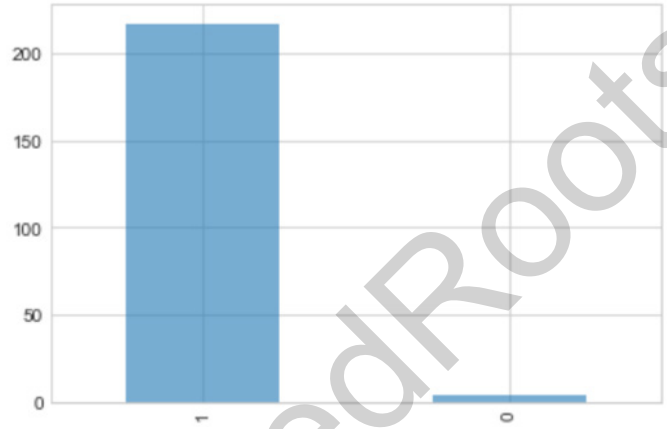
```
params = dict(eps = 0.195, min_samples = 24, metric = 'cosine')
model = DBSCAN(**params).fit(df_X_std.values)
```

```
df = df_cancer.assign(clusters = model.labels_).iloc[:, -2:]

df[df.clusters == 0].labels.value_counts().plot.bar(alpha = 0.6)
```



As can be inferred from the plot above, all the points clustered as non cancerous are actually non cancerous points

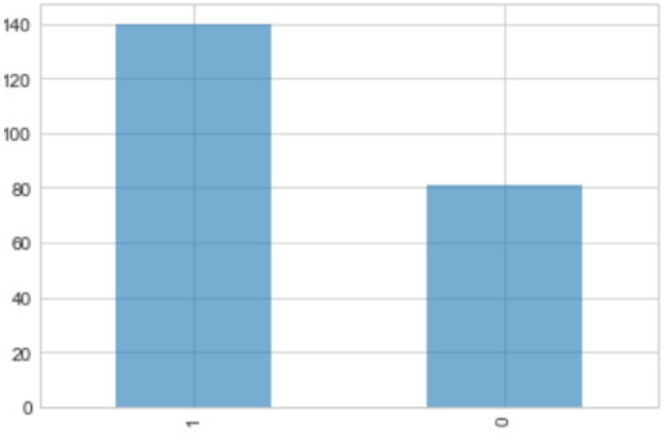
```
df[df.clusters == 1].labels.value_counts().plot.bar(alpha = 0.6)
```



As can be inferred from the plot above, almost all the points clustered as cancerous are actually cancerous points. A small fraction of non cancerous points were also clustered as cancerous.

Scikit Learn's DBSCAN clustering model labels noise points as negative one (-1). We can check the distribution of the actual labels within the points labelled as noise, as shown in the code below.

```
df[df.clusters == -1].labels.value_counts().plot.bar(alpha = 0.6)
```



From the above plot we can infer that the model considers around 140 cancerous points and 80 non cancerous points as noise.

ADVANTAGES AND LIMITATIONS OF DBSCAN CLUSTERING:

One of the most important advantages of DBSCAN is that the algorithm is resilient to noise/ outliers. It can also handle data that have clusters of different shapes and sizes - even non linearly separated clusters. On the other hand it requires that the clusters within the data be of uniform density. In cases where this requirement is not satisfied, the algorithm becomes sensitive to even small variations in its hyperparameters. The algorithm is not suited for handling high dimensional data especially if the distance measure specified is euclidean.

10. CONTENT BASED RECOMMENDER SYSTEM

CONTENT BASED RECOMMENDER SYSTEM

Recommender systems have become a mandatory feature for most modern online shopping portals and have been known to boost sales by orders of magnitudes, just by providing the customer with alternative choices while shopping. Today recommender systems are available for all kinds of products like books, clothing, music, movies, food, hotels, social network friends and the scope keeps widening.

Recommender systems can be classified into two basic types based on the technique used to provide the required recommendations. They are:

1. Content based recommender systems.
2. Collaborative Filtering based recommender systems.

CONTENT BASED RECOMMENDER SYSTEMS:

Content based recommender systems make use of the properties/attributes/metadata of the products previously chosen by the customer to recommend other similar products. It does this by vectorizing text, image and any other kind of data available, that describe all the available products. Then based on the customer's previous choices, it computes a similarity measure between the vectors of these previously chosen products, with the vectors of all other products available and then recommends top "n" products that have high vector similarity values.

Though the principle behind functioning of content based recommender systems is quite simple, it has its own drawbacks, one issue that arises is that of excessive specialization. In other words, given that a particular user purchases items from a particular set of categories, the system is not able to recommend items outside those categories, even though they could be interesting to them.

COLLABORATIVE FILTERING BASED RECOMMENDER SYSTEMS:

Collaborative filtering is currently the more frequently used approach and it usually provides better results than content based recommender systems. Some examples of this are found in the recommendation systems of Youtube, Netflix, and Spotify.

These kinds of systems utilize user - item interactions as the basis for their recommendations. The user - item interactions can be visualized using a matrix where every cell represents the interaction between an user and item. This results in a sparse matrix, a simplistic version of which is shown below:

	item_1	item_2	item_3	item_4
user_1	9	8	9	3
user_2	?	?	5	7
user_3	8	?	8	5

The above matrix represents the ratings from 1 to 10 given by the users to items. Given the similarity in the tastes of user_1 and user_3, it would be a good assumption that user_3 would like item_2. This is the fundamental principle behind Collaborative Filtering, which involves using matrix factorization methods to compute the most plausible ratings for the missing values in the user - item matrix.

In this chapter will be primarily exploring content based recommender systems.

CONTENT BASED RECOMMENDER SYSTEM FOR AMAZON APPAREL DATASET:

In the case of the image processing discussed previously in the chapter on SVD, we dimensionally reduced a dataset of images of a certain kind (ie: Faces). In the case of applying SVD on text data, we will use a dataset of text documents that pertain to a certain topic.

Consider the case of a dataset consisting of product descriptions, given by their corresponding manufacturers, on an online shopping portal. The basic technical parlance for text processing is as follows:

- The collection of all the text documents (ie: Product descriptions) would be called a **Corpus**.
- Each text document (product description) will simply be called a **Document**.
- The set of all the words contained in the Corpus is called its **Vocabulary**.

In the case of image processing, each image was vectorized by flattening its pixel matrix and then considering each pixel as a feature. In the case of text data, we wish to represent each document as a vectors using the following previously discussed techniques:

1. Bag of words (**BOWs**).
2. Term Frequency inverse document frequency (**TFIDF**)
3. Word 2 Vec (**W2V**).

THE AMAZON APPAREL DATASET:

For the sake of demonstration we will use the Amazon Apparel dataset, which basically consists of textual description of the clothing sold at Amazon along with the corresponding product images. We will implement an elementary recommendation system using the image and text vectorization techniques discussed in the previous chapters.

The amazon dataset consists of around 1.8 lakh entries consisting of eighteen features (columns). Most of these features have invalid or null entries as can be in the image below:

```
df1= pd.read_csv(data_path + 'amazon_apparel.csv')

df1.info()
```

```
Int64Index: 12128 entries, 46 to 183120
Data columns (total 19 columns):
 #   Column             Non-Null Count  Dtype
---  ------             --------------  -----
 0   sku                2 non-null      object
 1   asin               12128 non-null  object
 2   product_type_name  12128 non-null  object
 3   formatted_price    12128 non-null  object
 4   author             0 non-null      float64
 5   color              12128 non-null  object
 6   brand              12113 non-null  object
 7   publisher          3336 non-null   object
 8   availability       10726 non-null  object
 9   reviews            12128 non-null  object
 10  large_image_url    12128 non-null  object
 11  availability_type  10740 non-null  object
 12  small_image_url    12128 non-null  object
 13  editorial_review   145 non-null    object
 14  title              12128 non-null  object
 15  model              2692 non-null   object
 16  medium_image_url   12128 non-null  object
 17  manufacturer       3336 non-null   object
 18  editorial_reivew   10888 non-null  object
```

Out of all the features present in the dataset, we are interested only in five features for the sake of our purposes. They are:

1. **Asin:** is basically an acronym for “amazon standard identification number”.
2. **Title:** Refers to the product description provided by the clothing manufacturers.
3. **Product_type_name:** is basically a description of the type of clothing. Eg: Shirt, skirt, etc.
4. **Brand:** refers to the name of the clothing manufacturer.
5. **Color:** refers to the color(s) of the clothing.

As can be observed all the above data are textual or string data. The images corresponding to each of the products are “jpeg” files having asin numbers as their names and are available in a separate folder named “**amz_apparel**”. Thus every asin number corresponds to a product image.

Shown below is a sampling of the text data under “title”:

```
1   df.sample(10).title.values
```

```
array(['MP Happiness T-Shirt - Magnolia Pearl',
       "Entro Women's Sleeveless Shag Top Dusty Blue Polyester Shirts Small",
       'God Bless My Softball Player Graphite Heather T-Shirt - Tees2urdoor - XL',
       'Worthington Grommet Trapeze Top Size PM',
       'Bar III Sleeveless Mock-Neck Swing Top Black Combo XXL',
       "Dsquared2 Women's Plaid Front Short Sleeve Polo Shirt US XS IT 38",
       "Allison Brittney Women's Printed Cap Sleeve Popover Top, Medium, Blue Print",
       "Coeur Tri Tank With Shelf Bra Women's Triathlon XS Ginko",
       "Side Stitch Women's Pocket-Front Solid Button Down Shirt Gray XS",
       'Womens Topshop Cold Shoulder Lace Top, Size 8 (6-8 US) - White'],
      dtype=object)
```

As can be seen from the above output, the text contains non alphanumeric string objects. Also the text is not in spoken or grammatically correct language. Instead it contains short descriptive phrases.

DATA PREPROCESSING:

Real world raw data is generally quite messy. It is found to be consisting of missing values, wrong entries and duplicated data. This is more so for textual data. Raw text data has to be first transformed into a more usable form before any kind of vectorization or machine learning can be performed on it. The code shown below transforms the raw data described earlier into a more usable format.

Code block A performs the following tasks:

1. Attaches the suffix ".jpeg" to all asin names. This will be helpful later on, when we want to retrieve images corresponding to specific products.
2. Converts all text to non capitalised format.

Code block B performs the following tasks:

1. Replace missing values with an empty string/space.
2. Drops all duplicate entries with respect to title.

Code blocks **C & D** together perform the following tasks:

1. Block C uses the regex library to define patterns (regex_pat_1 & regex_pat_2), which need to be removed from the specified textual data:
   - Regex_pat_1 is a pattern that matches any non alphanumeric string data.
   - Regex_pat_2 is a pattern that matches any excess white space in string data.

2. Block D removes any textual data that match the patterns described in point 1.

```
img = df.asin + '.jpeg'
title = df.title.str.lower()
item = df.product_type_name.str.lower()
brand = df.brand.str.lower()
color = df.color.str.lower()

df_data = pd.DataFrame().assign(img = img, title = title, item = item, brand = brand, color = color)
```

A

```
df_data.fillna(' ', inplace = True)
df_data.drop_duplicates(['title'], inplace = True)
```

B

```
regex_pat_1 = re.compile(r'[^a-zA-Z0-9\n]') #--Matches all chars that are not alpha numeric
regex_pat_2 = re.compile(r'\s+') #------------Matches all excess spaces eg: '  '
```

C

```
df_data.title = df_data.title.str.replace(regex_pat_1, ' ').str.replace(regex_pat_2, ' ')
df_data.item = df_data.item.str.replace(regex_pat_1, ' ').str.replace(regex_pat_2, ' ')
df_data.brand = df_data.brand.str.replace(regex_pat_1, ' ').str.replace(regex_pat_2, ' ')
df_data_color = df_data.color.str.replace(regex_pat_1, ' ').str.replace(regex_pat_2, ' ')
```

D

The result of applying the code shown above, is that we transform the raw data to the format shown below:

```
df_data.sample(5)
```

	img	title	item	brand	color
8093	B06Y3KWH1P.jpeg	pol clothing women s sleeveless eyelash lace t...	shirt	pol clothing	grey
5017	B06XSVGZBN.jpeg	avia women s long sleeve performance t shirt w...	shirt	avia	beach glass/nine iron
5475	B01EXXFAP4.jpeg	white stag maternity short sleeve henley w poc...	apparel	white stag	white
4156	B071XD2D38.jpeg	dee elly womens 4si3nna off the shoulder ruffl...	shirt	dee elly	white
4504	B01DTFPENU.jpeg	carincci women s colombian shaping slimming to...	shirt	carincci intimates	white

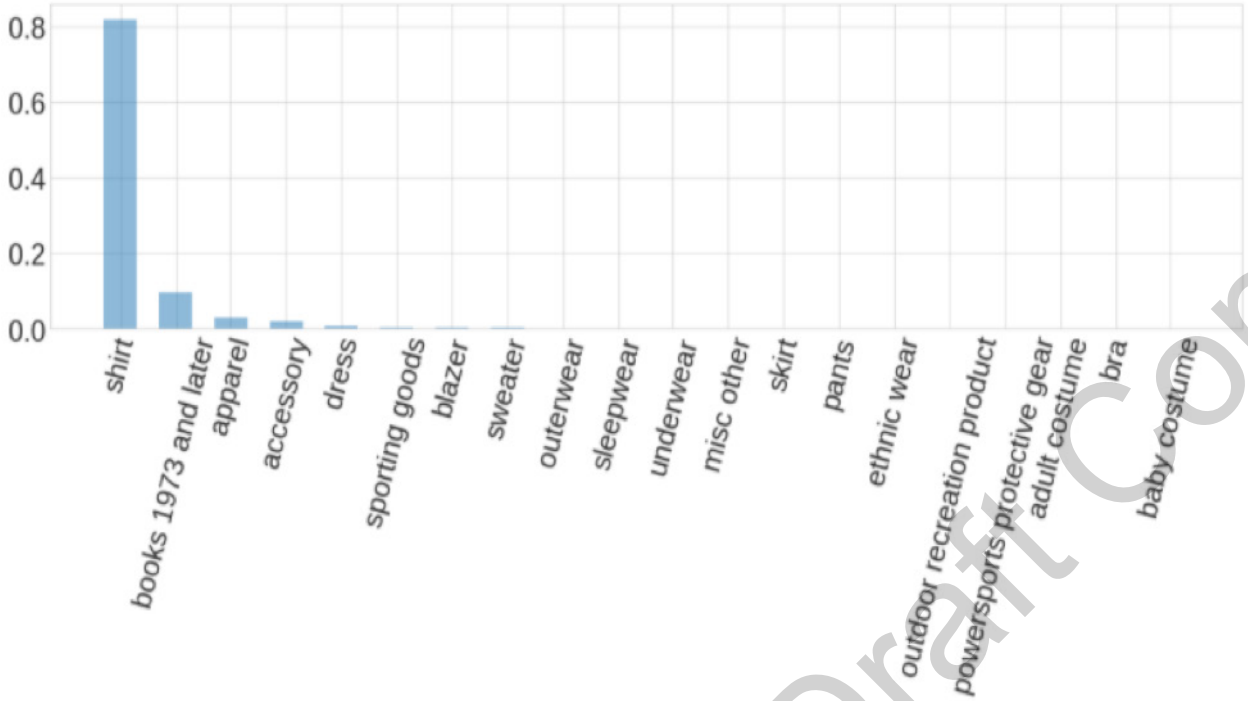
FEATURE STATISTICS:

Shown below are bar plots that show the proportions of the various attributes of the products:

A. ITEM (PRODUCT TYPE):

```
df, col_name = df_data, 'item'

df_prod_propotions = fn_plot_proprotions(df, col_name)
```

```
TOP 20 of 30 CATEGORIES:
```



As can be seen from the plot above, more than 80% of the clothing items are listed under "shirt". The entry "books 1973 and later" does not make sense. The item types listed as "apparel" and "dress" seem too general for item description.

The items listed as "books 1973 and later" is the second largest among all the items listed and hence it could impact the vectorization process if not corrected. We investigate it as shown below:

```
df_data[df_data.item == 'books 1973 and later'].title.sample(10).values
```

```
array(['fashion women s cornell university cu logo tee black size xxl',
       'togeb impractical jokers logo v neck short t shirt young t shirts black xl',
       'qincent women pre cotton tshirt unicycle hockey',
       'ama tm women christmas deer printed casual splicing blouse tops dress x large red ',
       'vansty i m a monkey cute with tail o neck t shirt for lady yellow size xl',
       'officially licensed merchandise the hulk i am the hulk girly t shirt d grey xx large',
       'dreamtravel womens latest arrival sleeveless lace shirts v neck cami tank tops beige medium',
       'dirty heads band cabin by the sea garland woman graphic print treasures crop tee',
       'unitato half unicorn half potato 3 4 sleeve raglan long sleeve',
       'farysays women s sexy cold shoulder blouse lace up ribbed tops green xxl'],
      dtype=object)
```

As can be seen from the above output, it seems that these items were named wrongly as "books 1973 and later". We check items listed as "dress" and "apparel" in the same manner:

```
df_data[df_data.item == 'dress'].title.sample(10).values
```

```
array(['women s off shoulder casual tops summer fashion stripe blouse shirt moonhouse',
       'our precious women s casual 3 4 sleeve scoop neck heart print tunic dress',
       'misaky women s fashion 2017 print long sleeve zippered back top blouse',
       'tifenny women casual long sleeved striped shirt loose blouse',
       'igenjun women s v neck short sleeve flare tunic tops',
       'levaca women s long sleeve button deco back pocket casual loose t shirt dress',
       'womens fashion oversized striped casual irregular maxi dress long sleeve loose fit',
       'feitong women casual sexy flare sleeve strapless tops blouse',
       'eva franco women s seymour top',
       'women s boho long sleeve loose swing dress casual dress 016'],
      dtype=object)
```

```
df_data[df_data.item == 'apparel'].title.sample(10).values
```

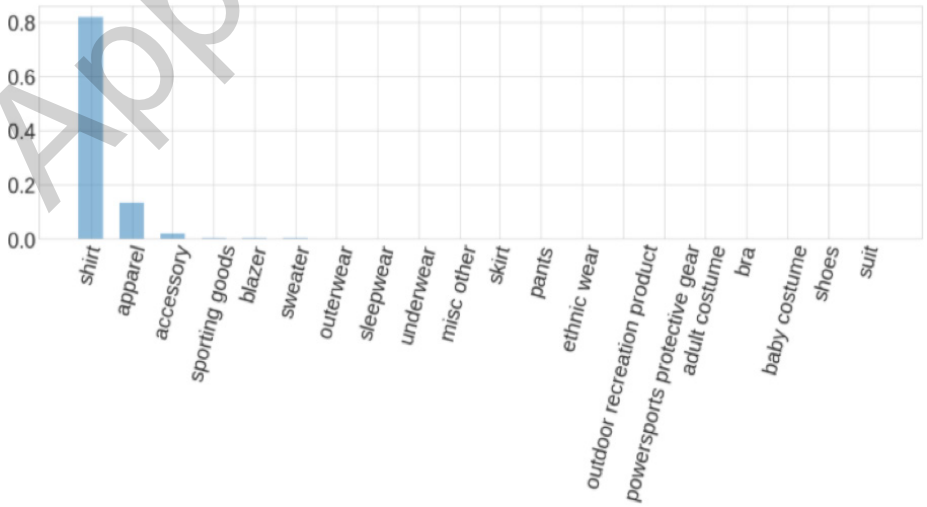
```
array(['social butterfly house women s ariana off shoulder denim crop top',
       'long sleeve v neck brown print cotton dress unique women s fashions',
       'nygard women s regular slims bohemian chic caftan',
       'winter kate sheer v neck kimono sleeve blouse x small black',
       'next level women s cvc crew',
       'autumn melody women fashion temperament round neck long sleeves shirt blouse tops',
       'august silk gray top blouse cap sleeve size l nwt movaz',
       'studio m long sleeve geo print slit neck blouse black white large',
       'social butterfly house women s sweet strings tank lavender purple',
       'kingfield women print irregular long sleeve loose casual cardigan'],
      dtype=object)
```

Items listed under "dress" and "apparel" have no significant difference and are similar to those listed under "books 1973 and later" and hence we shall merge these three categories into one, under "apparel" as shown below:

```
a, b = 'books 1973 and later', 'dress'
df_data.item = df_data.item.str.replace(a, 'apparel').str.replace(b, 'apparel')
```

```
df_, col_name = df_data, 'item'

df_prod_propotions = fn_plot_proprotions(df_, col_name)
```

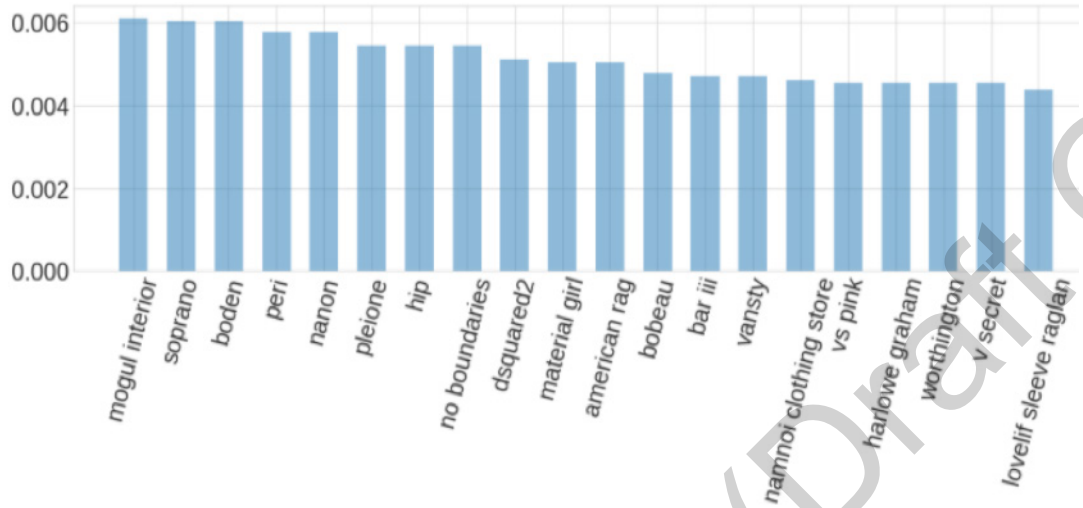
```
TOP 20 of 28 CATEGORIES:
```



B. BRAND:

```
1  df_, col_name = df_data, 'brand'
2
3  df_prod_propotions = fn_plot_proprotions(df_, col_name)
```

```
TOP 20 of 2744 CATEGORIES:
```

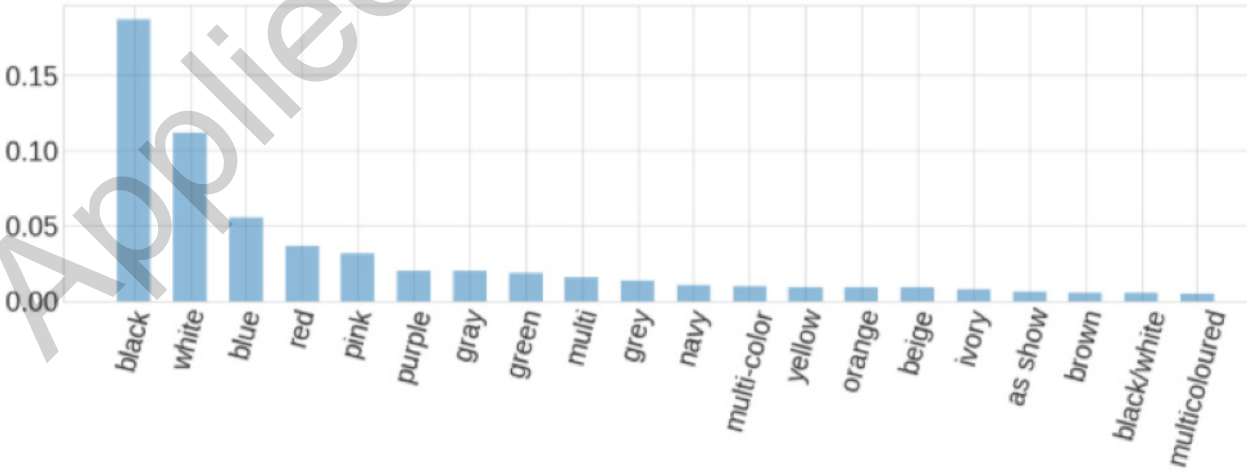


As can be seen from the above plot, the brands of the clothing seems to be quite evenly distributed.

C. COLOR:

```
1  df_, col_name = df_data, 'color'
2
3  df_prod_propotions = fn_plot_proprotions(df_, col_name)
```

```
TOP 20 of 2810 CATEGORIES:
```



From the plot above we infer that most clothing are colored black, followed by white, blue, red and pink. This concludes the data preprocessing and exploration stage.

CONTENT BASED RECOMMENDER SYSTEM:

The process for creating a basic content based recommender system for the clothing available on amazon, will consists of the following steps:

1. Image vectorization.
2. Text Vectorization.
3. Recommendation based on image/text similarity.

1. IMAGE VECTORIZATION:

The images of the clothing in the amazon apparel dataframe created previously (**df_data**) are stored in the "amz_apparel" folder. The size of these images are 224 x 224 x 3, which means they are stored as RGB colored images. The output of the code below confirms this.

```
[fn_img_shape(np.random.randint(0, 10_000)) for i in range(5)]
```

```
[(224, 224, 3), (224, 224, 3), (224, 224, 3), (224, 224, 3), (224, 224, 3)]
```

Shown below are some samples of images:

```
list0_imgs = df_data.sample(6).img.values

fn_show_images(list0_imgs, img_path)
```

We first covert each of the images in the amz_apparel folder into vectors by first converting the images into grayscale pixel matrices and then flattening these matrices to form single vectors as shown in the code below. Note that we use the **cv2** python library for this purpose. Thus we end up with 12128 vectors each of length 50176, stored in matrix format.

```
def fn_create_array0_all_imgs(list0_img_names, img_path):

    list0_img_arrys = []
    pbar = ProgressBar()
    for img_name in pbar(list0_img_names):

        img = cv2.imread(img_path + img_name, 0) # image read as grey_scale
        img_arry = np.asarray(img, dtype = 'uint8').flatten()
        list0_img_arrys.append(img_arry)

        array0_img_arrys = np.array(list0_img_arrys)

    return array0_img_arrys
```

```
img_path = data_path + 'apparel_imgs/'
list0_img_names = df_data.img.values

array0_imgs = fn_create_array0_all_imgs(list0_img_names, img_path)
array0_imgs.shape
```

```
100% (12128 of 12128) |###################| Elapsed Time: 0:16:36 Time:  0:16:36
(12128, 50176)
```

We then use the scikit learn library to perform SVD on the above matrix and perform dimensionality reduction from 50176 dimensions to 500 dimensions as shown in the code below. Note that scikit learn ingests data in tabular format instead of the linear algebraic format.

```
from sklearn.decomposition import TruncatedSVD
array0_imgs = array0_imgs/255.0 #--------- NORMALIZE

svd = TruncatedSVD(500)
%time array0_encoded_images = svd.fit_transform(array0_imgs)

array0_encoded_images.shape
```

```
CPU times: user 6min 43s, sys: 8.73 s, total: 6min 52s
Wall time: 3min 33s
(12128, 500)
```

2. TEXT VECTORIZATION USING BOW AND TFIDF:

As discussed in the previous chapter, Bag of words and Term frequency Inverse Document Frequency techniques are count based techniques. These can be implemented using the scikit learn library. The image below describes the BOW class (ie: CountVectorizer) and the default values of some its most relevant parameters:

```
[1]: vectorizer = CountVectorizer(stop_words = None,
                                  ngram_range = (1, 1),
                                  max_df = 1.0,
                                  min_df = 1,
                                  binary = False)
```

Before we can perform text vectorization, we combine all the text features (title, item, brand and color) into one consolidated format as shown below:

```
def fn_set(row):
    row = str(row)
    list0_words = row.split()
    set0_words = set(list0_words)
    row = ' '.join(list(set0_words))
    return row
```

```
set0_brands = set(df_data.brand)
set0_colors = set(df_data.color)
set0_prod_type = set(df_data.item)

len(set0_brands), len(set0_colors), len(set0_prod_type)
```

```
(2744, 2810, 28)
```

```
text = df_data.title + ' ' +  df_data.brand +  '  ' + df_data.color +  '  ' + df_data.item
df_img_text = pd.DataFrame().assign(img = df_data.img, text = text)

df_img_text.text = df_img_text.text.apply(fn_set)
df_img_text.head()
```

	img	text
0	B01NACPBG2.jpeg	junior graphic sleeve black shirt tees l foil ...
1	B011YPK0MW.jpeg	dip navy size one shirt dye the piece top in s...
2	B011JQWCCM.jpeg	shield xl ferrari shirt puma rosso women big r...
3	B01NAZ3L3C.jpeg	3 rhinestone diamonds and cross sleeve black s...
4	B01I5GRO18.jpeg	blouse fifteen eggshell shirred hem ruffle shi...

The text shown in the data frame **df_img_text** shown above is now ready for vectorization since it contains within it all textual information that is product descriptive.

BOW CODE:

Shown below is the code to create BOW vectors using scikit learn's CountVectorizer class:

```
corpus = df_img_text.text.values

%time BOW_vectorizer = CountVectorizer(stop_words = 'english').fit(corpus)
bow_matrix = BOW_vectorizer.transform(corpus)

bow_matrix.shape
```

```
CPU times: user 157 ms, sys: 0 ns, total: 157 ms
Wall time: 159 ms
(12128, 9775)
```

The implementation of the code above gives us a scipy.sparse.csr_matrix (**bow_matrix**). The scipy sparse matrix format is a compressed format for sparse matrices, which stores the values in terms of non-zero values and their corresponding matrix index location (**non_zero_value : location**) pairs, instead of the entire matrix structure itself. Since most of the values are zero, this format achieves a high degree of data compression. Note that we have performed **stop word** removal on the text prior to the vectorization, by setting the stop_words parameter in the CountVectorizer instance above to '**english**'.

As can be seen from the code's output, the shape of **bow_matrix** is 12128 x 9775 indicating that there are 9775 words in the vocabulary and thus there are that many features for each of the 12128 vectorized documents.

We then apply SVD on the above matrix to obtain a dimensionality reduction from 9554 to 500 dimensions. This is done as shown in the code below:

```
svd = TruncatedSVD(500)
%time dim_reduced_bow_matrix = svd.fit_transform(bow_matrix)

dim_reduced_bow_matrix.shape
```

```
CPU times: user 10.4 s, sys: 3.21 s, total: 13.6 s
Wall time: 7.54 s
(12128, 500)
```

The final matrix named **dim_reduced_bow_matrix** will be used in the final stage during product recommendation.

TF IDF CODE:

The process for performing TF IDF text vectorization is similar to that used for BOW. The code for implementing it is shown below, which gives us a sparse matrix - **tfidf_matrix**. Just as previously done for BOW, we apply SVD on the sparse matrix got from the TF IDF process to get our final 500 _D dimensional reduced text vectors in the form of **dim_reduced_tfidf_matrix**:

```
corpus = df_img_text.text.values

TFIDF_vectorizer = TfidfVectorizer(stop_words = 'english').fit(corpus)
tfidf_matrix = TFIDF_vectorizer.transform(corpus)

tfidf_matrix.shape
```

```
(12128, 9775)
```

```
svd = TruncatedSVD(500)
%time dim_reduced_tfidf_matrix = svd.fit_transform(tfidf_matrix)

dim_reduced_tfidf_matrix.shape
```

```
CPU times: user 10 s, sys: 2.91 s, total: 12.9 s
Wall time: 7.25 s
(12128, 500)
```

RECOMMENDATIONS (COMPUTING VECTOR SIMILARITY):

GIven a prior preference, content based recommendation is basically about returning back the top 'n' products with most similar vectors to the vector of the prior preference. Shown below is code that implements this based on euclidean distance.

```
def fn_similar_vecs(query_idx, df_img_text, matrix0_vecs,
                    n_recommendations = 4, buffer = 2):

    query_img  = df_img_text.img.values[query_idx]
    list0_losses = []
    for idx in range(len(matrix0_vecs)):

        # EUCLEDIAN DIST (ie: LOSS):
        loss = (((matrix0_vecs[query_idx] - matrix0_vecs[idx])**2).sum())**(1/2)
        list0_losses.append(loss)

    arry0_losses = np.array(list0_losses)
    best_idxs = np.argsort(arry0_losses)[1:n_recommendations+buffer]
    list0_most_simi_imgs = df_img_text.iloc[best_idxs].img.values
    return list0_most_simi_imgs
```

The parameter "matrix0_vecs" in the function above refers to any of the text or image matrices computed earlier. Shown below are some example recommendations:

```
query_idx = 999

list0_imgs_image_based = fn_similar_vecs(query_idx, df_img_text, dim_reduced_img_matrix)
list0_imgs_bow_based = fn_similar_vecs(query_idx, df_img_text, dim_reduced_bow_matrix)
list0_imgs_tfidf_based = fn_similar_vecs(query_idx, df_img_text, dim_reduced_tfidf_matrix)

fn_show_imgs(query_idx, list0_imgs_image_based, list0_imgs_bow_based, list0_imgs_tfidf_based)
```

QUERY:

RECOMMENDATIONS


```
query_idx = 1212

list0_imgs_image_based = fn_similar_vecs(query_idx, df_img_text, dim_reduced_img_matrix)
list0_imgs_bow_based = fn_similar_vecs(query_idx, df_img_text, dim_reduced_bow_matrix)
list0_imgs_tfidf_based = fn_similar_vecs(query_idx, df_img_text, dim_reduced_tfidf_matrix)

fn_show_imgs(query_idx, list0_imgs_image_based, list0_imgs_bow_based, list0_imgs_tfidf_based)
```

QUERY:

RECOMMENDATIONS:

THE W2V BASED RECOMMENDATION:

The w2v model we will be using for creating **TFIDF_weighted_w2v** word vectors of the words in amazon apparel corpus, will be a compressed version of the freely available google news w2v pretrained vectors. The google news w2v model is basically a **dictionary** having various words in the english language as its **keys** and the **values** being the corresponding 300-D word vector. As can be seen from the output of the code below, the w2v model we will be using is a dictionary with just 46603 word and word_vector key-value pairs, chosen such that they are more than sufficient for processing the amazon_apparel corpus.

```
with open(data_path + 'word2vec_model', 'rb') as handle:
    w2v_model = pickle.load(handle)  #-------------------------Dict of {word:vector}
```

```
len(w2v_model)
```

```
46603
```

We will use the function **fn_tfidf_weighed_w2v_doc_vecs** shown below for creating the required w2v based document vectors. The function basically contains the following three main code blocks.

A. This code block is a function that takes in two parameters: a word and its corresponding TFIDF value (ie: weight) and returns the w2v word vector of that word scaled by the TFIDF value. It is this function that makes use of the w2v model/dictionary. Note that fn_w2v returns a 300-D zero vector if the word vector corresponding to any particular word is not available in the w2v_model/dictionary.

B. This code block uses the document vectors iterated from the TFIDF matrix and does the following:

   a. Identify indexes that correspond to non zero values in the current row vector being iterated. These indexes correspond to words present in that document.

   b. Creates two lists/arrays - one containing the actual words in the document corresponding to that particular row of the TFIDF matrix (**doc_words**) and the other containing the corresponding TF IDF values of the words in that document (**tfidf_row**).

C. This code block simultaneously iterates over the **doc_words** and **tfidf_row** and uses **fn_w2v** to compute the TFIDF-weighted_w2v documents vectors for every row in the TFIDF matrix.

```
def fn_tfidf_weighed_w2c_doc_vecs(tfidf_matrix, bow_matrix, tfidf_features, w2v_model):

    pbar = ProgressBar(max_value = tfidf_matrix.shape[0])
    listO_weighed_w2c_doc_vecs = []
    def fn_w2v(word, weight): return list(w2v_model.get(word, np.zeros(300)) * weight)

    for tfidf_row, bow_row in pbar(zip(tfidf_matrix, bow_matrix)):

        non_zero_idxs = tfidf_row.toarray().squeeze().nonzero()[0]
        tfidf_row = tfidf_row.toarray().squeeze()[non_zero_idxs]
        doc_words = np.array(tfidf_features)[non_zero_idxs]

        iter = zip(doc_words, tfidf_row)
        weighed_word_vecs = np.array([fn_w2v(word, weight) for word, weight in iter])
        weighed_w2c_doc_vecs = sum(weighed_word_vecs)/(sum(tfidf_row) + 1e-5)

        listO_weighed_w2c_doc_vecs.append(list(weighed_w2c_doc_vecs))
        pbar.update()

    arryO_weighed_w2c_doc_vecs = np.array(listO_weighed_w2c_doc_vecs)
    return arryO_weighed_w2c_doc_vecs
```

A

B C

Avoid zero division error

```
arry0_w2v_doc_vecs = fn_tfidf_weighed_w2c_doc_vecs(tfidf_matrix, bow_matrix,
                                                   tfidf_features, w2v_model)

arry0_w2v_doc_vecs.shape
```

```
100% (12128 of 12128) |###################| Elapsed Time: 0:00:21 Time:  0:00:21
(12128, 300)
```

W2V RECOMMENDATIONS:

Just as done previously, recommendations are simply the top "**n**" most similar vectors (w2v document vectors)with respect to the vector of document that has been currently selected. Show below are some example recommendations and their comparison to the image based recommendations.

```
query_idx = 10999

list0_imgs_image_based = fn_similar_vecs(query_idx, df_img_text, dim_reduced_img_matrix)
list0_imgs_tfidf_based = fn_similar_vecs(query_idx, df_img_text, arry0_w2v_doc_vecs)

fn_show_imgs(query_idx, list0_imgs_image_based, list0_imgs_tfidf_based)
```

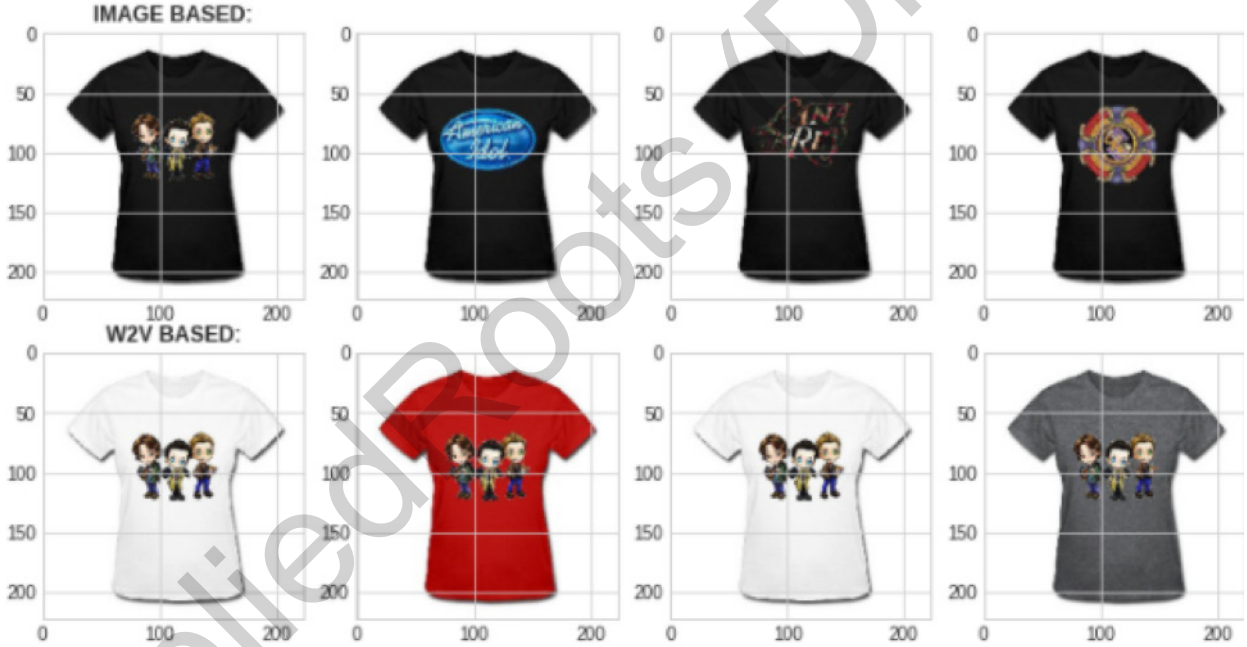
QUERY:

RECOMMENDATIONS:

```
query_idx = 80

list0_imgs_image_based = fn_similar_vecs(query_idx, df_img_text, dim_reduced_img_matrix)
list0_imgs_tfidf_based = fn_similar_vecs(query_idx, df_img_text, arry0_w2v_doc_vecs)

fn_show_imgs(query_idx, list0_imgs_image_based, list0_imgs_tfidf_based)
```

QUERY:

RECOMMENDATIONS:



The recommendations shown above are interesting as it shows outcomes from using different similarity perspectives: Image similarity and document vector similarity. Note that:

1. Image similarity produces shirts of the same type and color with different symbols in the middle
2. w2v_document similarity produces shirts of the same type, with the same type of symbol at the centre, but in different colors.

It is good practice to use both visual and the textual data within recommender systems so as to make the recommendations more broader in its scope of recommendations.

11. COLLABORATIVE FILTERING BASED RECOMMENDERS

(NON NEGATIVE MATRIX FACTORIZATION)

MATRIX FACTORIZATION – 2
(NON NEGATIVE MATRIX FACTORIZATION)

Matrix factorization refers to the linear algebraic technique of decomposing matrices into the product of two or more other matrices . In other words it involves mathematically approximating "factor matrices", the product of which creates the most closest reconstruction of the original matrix.

There is a set of mathematically proved, matrix decompositions rules in linear algebra, two of which are quite extensively used in unsupervised machine learning, they are:

1. Singular Value Decomposition (SVD)
2. Non negative Matrix Factorization (NMF)

LATENT VARIABLES (SINGULAR VALUE DECOMPOSITION REVISITED):

As discussed previously, matrices can be considered as:

1. Collection of vectors (A dataset with observations (vectors) expressed in terms of their features.
2. Collection of axes of a specific feature space (transformation matrix - each column is an axis).
3. They could also be considered as representing some relationship between two interacting concepts.

The first two types of matrices have already been discussed in detail. This chapter will be basically about the third type of matrix mentioned above.

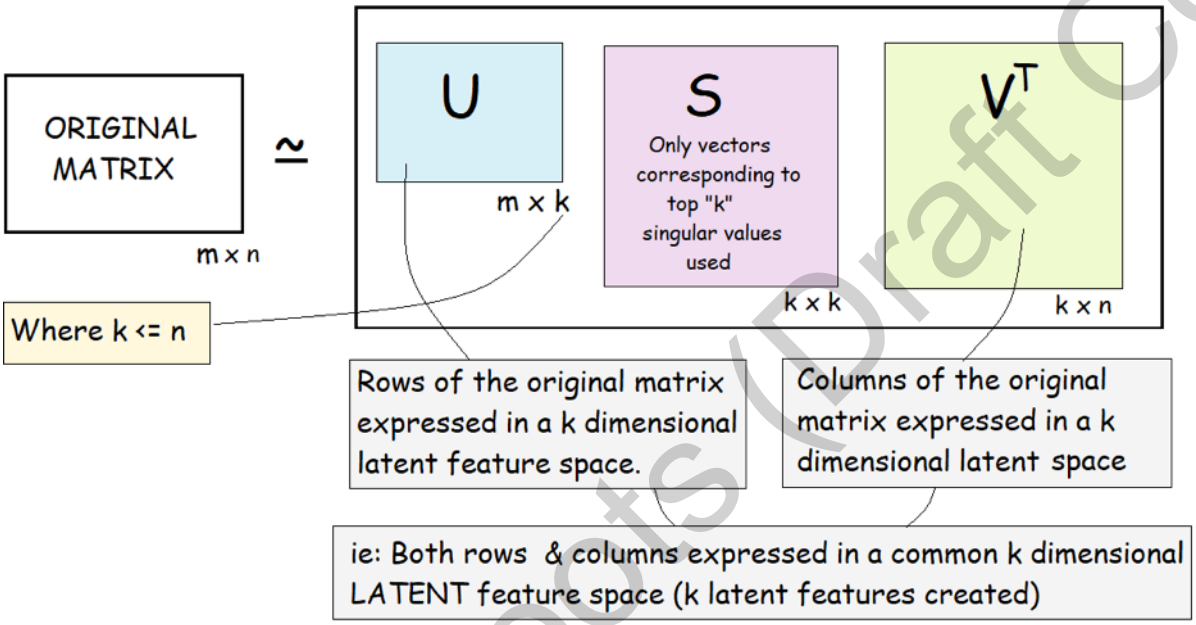
Consider the **TFIDF** matrix representation of a corpus. Each cell of the **TFIDF** matrix represents a relationship between a word and a document. Depending on how one wishes to see it, the TFIDF matrix could be seen as:

1. A matrix representing the words in terms of the documents in the corpus. Here each word is considered as a vector and each document as a particular feature.
2. A matrix representing the documents in terms of the words in the corpus. Here each document is considered as a vector and each word as a particular feature.

Note that we do not have an independent vector representation of either the words or the documents. In other words, we can only express only one of the following:

1. Words as vectors and in "document space".
2. Documents as vectors in "word space".

Singular Value Decomposition can be used to create latent variables from such kinds of "**relationship matrices**". (Latent means hidden or not obviously apparent). In other words, SVD can be used to create a common latent feature space. This latent feature space would be such that, each of the row components (ie: words) and the column components (ie: documents) of the matrix can be expressed as unique independent vectors within it. To be more specific, each row component and each column component can be expressed in a **common feature space** as unique row vectors and unique column vectors. This is further expressed in the image shown below:

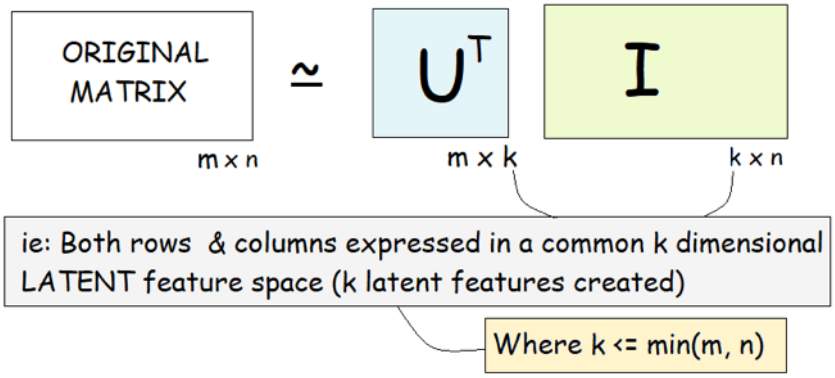


Thus the matrix **V** shown above could be considered as an alternate representation of the column vectors in the original matrix, but containing its own unique **k** dimensional vectors. Similarly the matrix U could be considered as an alternate representation of the row vectors of the original matrix.

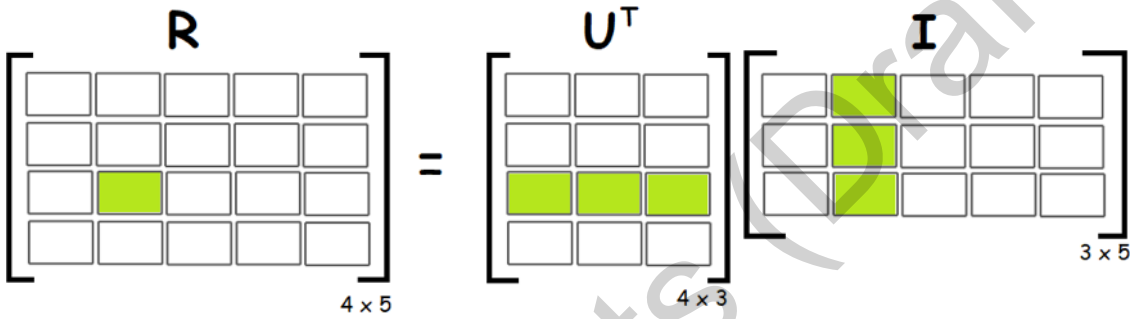
NON NEGATIVE MATRIX FACTORIZATION (NMF):

As previously discussed in the chapter on Singular Value Decomposition, SVD views the original matrix as the sum of 'k' individual matrices having the same size as that of the original matrix. Each of these individual component matrices is the result of the outer dot product of a vector from the U matrix and a corresponding vector from the V matrix.

The Non Negative Matrix Factorization technique has its own unique way of factoring matrices and it is only applicable to matrices which contain only non negative values (ie: zero or above). Unlike SVD, non-negative factorization reconstructs the original matrix cell by cell. It decomposes the original non negative matrix into two non negative "factor matrices" in the manner described in the image below:



The value in each cell of the original matrix is the outcome of the dot product between a row vector belonging to matrix $\mathbf{U_T}$ and column vector belonging to matrix **I** as shown in the image below:

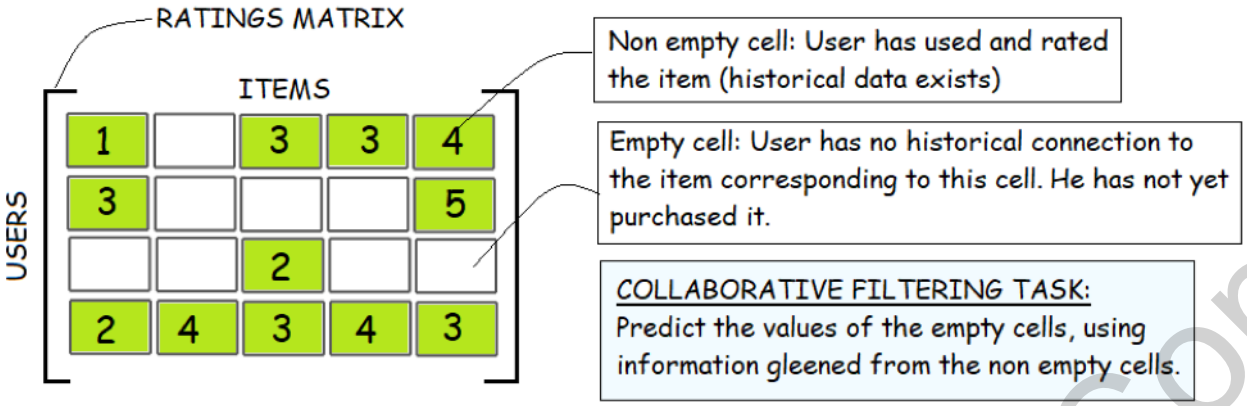


Every **cell(i, j)** is reconstructed as the dot product of a unique pair of vectors - A **row_vector_i** of matrix **UT** and a **column_vector_j** of matrix **I**. Note that the maximum value of k is constrained to be lesser than or equal to the smaller dimension of the original matrix **R**, In the example shown above, the value of **k** cannot be more than four.

This attribute of NMF, where the matrix reconstruction happens on the individual cell level, makes it uniquely suited for the matrix approximation technique called Collaborative Filtering, which is used extensively in recommender systems.

COLLABORATIVE FILTERING USING NMF:

Collaborative Filtering is at its core a recommendation technique that uses the past product ratings and reviews of a large enough mass of customers and then provides recommendations for alternate products not yet explored by each customer. In other words, the problem that collaborative filtering aims at solving is this: Given a matrix representing the user - item relationship of a particular shopping portal, predict the values for those cells in the matrix that represent user - item interactions that have not yet occured. This is further expressed in the image below:



The core idea that collaborative filtering aims to implement is that two users having similar or close ratings for products common to both are very likely to have similar ratings for products that are not common to both. In other words, people who tend to agree on most subjects, will most likely tend to agree on subjects that they have not yet explored. In the user-item matrix shown above, the users corresponding to row_0 and row_3 are quite similar and so it is quite probable that user_0 will like the item corresponding to column_1.

Recommendation systems generally have user-item matrices that are extremely huge (millions or rows and columns) with high sparsity (containing many empty values). For collaborative filtering tasks that need to scale to such large sizes, NMF coupled with gradient descent optimization has been used quite effectively. The objective function for such a task is defined as shown below:

$$U^*I^* = \underset{U,I}{\operatorname{argmin}} \sum_{\forall R_{i,j} \neq 0} \left(R_{i,j} - U_i^T I_j\right)^2$$

The above objective function basically says: " Use all the data in the non zero cells to predict values for the empty cells" . In other words, given a sparse user-item matrix **R**, find the best possible matrix pair **U** and **I** such that their matrix product of $\mathbf{U^TI}$ creates another matrix $\hat{\mathbf{R}}$ which:

1. Most closely approximates the non empty cells of **R** (minimum squared error).
2. Fills the empty cells of **R** with the most probable values.

Basically collaborative filtering is matrix factorization using lesser constraints. The original matrix **R** is an incomplete matrix to start with. The matrix factorization is performed such that:

1. All the values in the non empty cells of matrix **R** are used to create a common latent space.
2. The users and items in **R** are represented in this latent space as unique vectors.

3. These **latent** user_vectors and item_vectors are such that their dot products fill each empty cell of **R**, while at the same time, best approximating the values of its non zero cells.

The more sparse matrix **R** is, the less constrained the matrix factorization becomes. To avoid overfitting, some form of regularization is applied over the individual user_vectors and _item_ vectors, thus yielding two regularization hyperparameters that can be tuned.

THE MOVIE LENS DATASET:

For demonstration purposes we will use the movie lens dataset. The dataset contains around 10,000 ratings given by around 600 users on 9500 movies. Apart from the ratings data, the data also comes along with another table that contains metadata about the movies.

Show below is are samples of the ratings data and the metadata:

A. RATINGS DATA:

```
df_ratings = pd.read_csv(data_path + 'df_ratings.csv')

df_ratings.sample(5)
```

	user_id	movie_id	rating	timestamp
67915	438	4487	3.5	1105763904
7377	50	138208	2.5	1514239906
77499	483	2617	3.0	1178293971
52015	339	150	4.0	1460182140
24546	169	6252	2.5	1059429354

B. MOVIES METADATA:

```
df_movies = pd.read_csv(data_path + 'df_movies.csv')

df_movies.head()
```

	movie_id	title	genre	year
0	1	Toy Story	Fantasy	1995
1	2	Jumanji	Fantasy	1995
2	3	Grumpier Old Men	Romance	1995
3	4	Waiting to Exhale	Romance	1995
4	5	Father of the Bride Part II	Comedy	1995

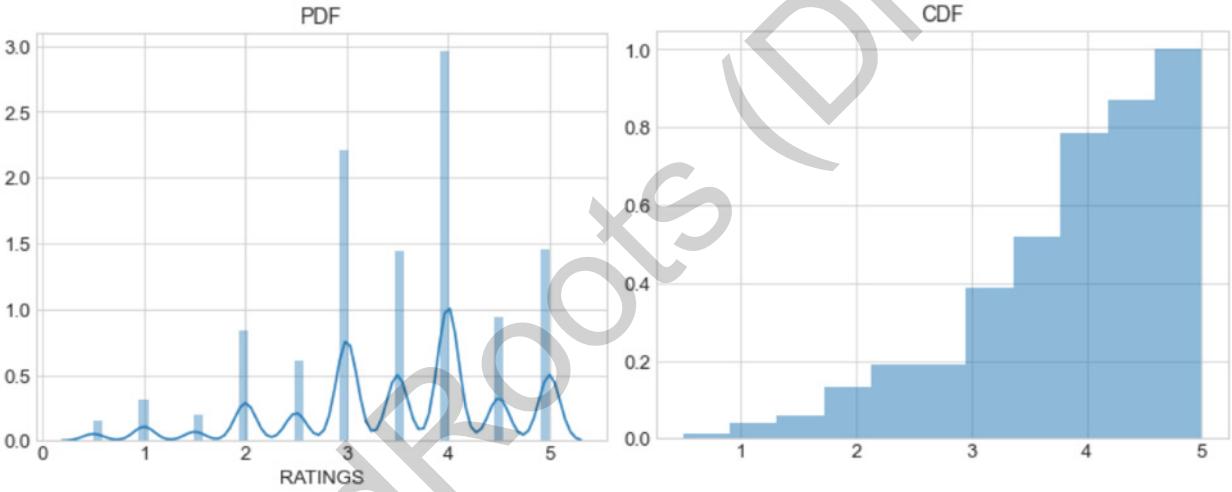
DATA EXPLORATION:

Given the two dataframes shown earlier, we will explore the data in the following ways:

1. Visualize the distribution of the movie ratings.
2. Visualize the average ratings of a movie and how widely they have been watched. (ie: avg_ratings Vs n_users).
3. Visualize the the rating trend with respect to time (years)
4. Visualize the distribution of movie genres.

1. DISTRIBUTION OF MOVIE RATINGS:

```
rand_var = df_ratings.rating.values
xlabel = 'RATINGS'

fn_show_distr(rand_var, xlabel)
```



As can be seen from the PDF and CDF plots above, most of the movies are rated in the range of 3 starts and above. The highest number of ratings are 4 stars.

The code below gives the rating values allowed:

```
sorted(df_ratings.rating.unique())
```

```
[0.5, 1.0, 1.5, 2.0, 2.5, 3.0, 3.5, 4.0, 4.5, 5.0]
```

We deduce the actual number of movies and users as shown below:

```
print(f'n_movies: {len(df_ratings.movie_id.unique())}')
print(f'n_users: {len(df_ratings.user_id.unique())}')
```

```
n_movies: 9724
n_users: 610
```

2. AVERAGE MOVIE RATING Vs NUMBER OF VIEWERS:

To plot the average movie ratings against the number of viewers per movie, we use the following code blocks explained/shown below:

A. This code block groups the dataframe "df_ratings" by the 'movie_id" column and counts the number of ratings per movie. It then presents this information as a dataframe with the following columns: **['movie_id', 'n_ratings']**

B. This code block groups the dataframe "df_ratings" by the 'movie_id" column and computes the mean value of the ratings per movie. It then presents this information as a dataframe with the following columns: **['movie_id', 'avg_rating']**

C. This code creates a dataframe called **"df_movies_meta"** which is the result of merging the **df_movies** data frame with the two dataframes created in code blocks **A** and **B**.

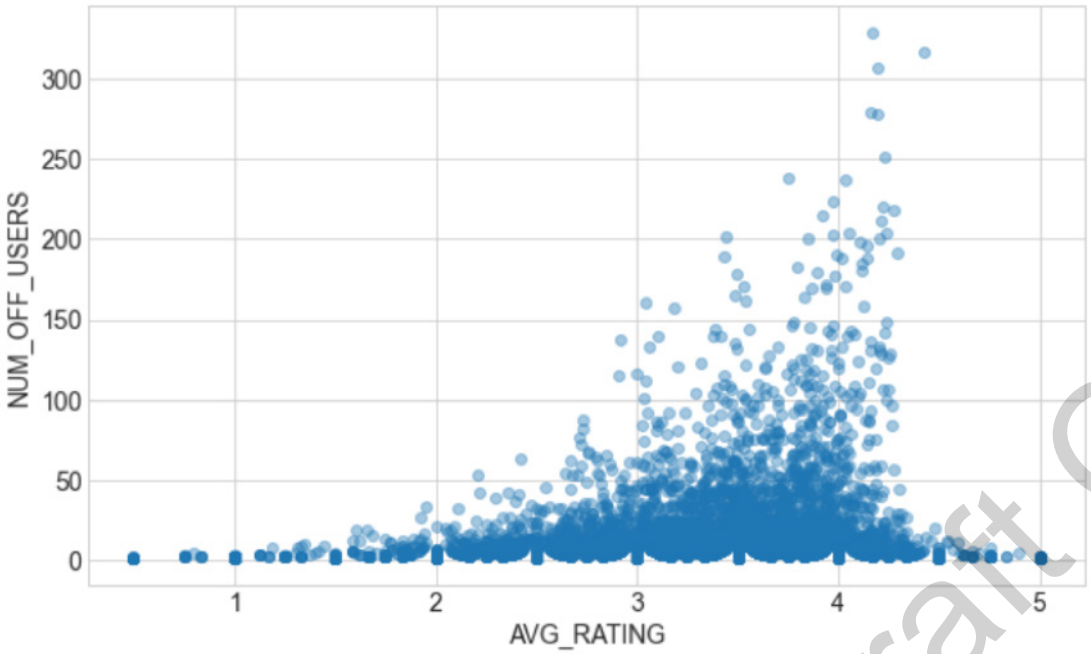
```
1  df_n_users = df_ratings.groupby('movie_id')['rating'].count()
2  df_n_users = df_n_users.to_frame().reset_index()
3  df_n_users.columns = ['movie_id', 'n_users']
4
5  df_avg_rating = df_ratings.groupby('movie_id')['rating'].mean()
6  df_avg_rating = df_avg_rating.to_frame().reset_index()
7  df_avg_rating.columns = ['movie_id', 'avg_rating']
8
9  df_movies_meta = df_movies.merge(df_n_users, on = 'movie_id')
10 df_movies_meta = df_movies_meta.merge(df_avg_rating, on = 'movie_id')
11
12 df_movies_meta.head()
```

A B C

	movie_id	title	genre	year	n_users	avg_rating
0	1	Toy Story	Fantasy	1995	215	3.920930
1	2	Jumanji	Fantasy	1995	110	3.431818
2	3	Grumpier Old Men	Romance	1995	52	3.259615
3	4	Waiting to Exhale	Romance	1995	7	2.357143
4	5	Father of the Bride Part II	Comedy	1995	49	3.071429

After arriving at the data frame shown above (df_movies_meta), we plot the scatter plot of the n_users and avr_rating columns of the dataframe as shown below:

```
1  X = df_movies_meta.avg_rating
2  Y = df_movies_meta.n_users
3  xlabel = 'AVG_RATING'
4  ylabel = 'NUM_OFF_USERS'
5
6  fn_scatter_plot(X, Y, xlabel = xlabel, ylabel = ylabel)
```



As can be seen from the above plot the most watched movies generally have ratings between 3 stars and 4.5 stars.

3. VISUALIZING THE RATINGS TREND WITH RESPECT TO TIME:

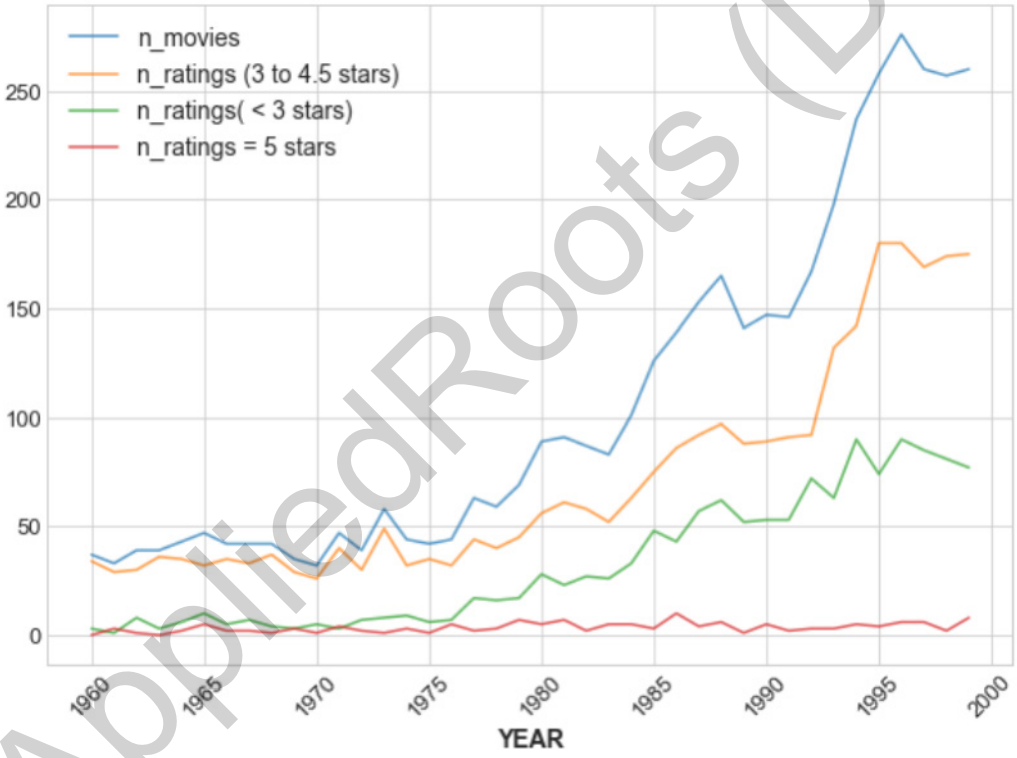
To be able to visualize the ratings trend, we first create the data frame shown below (**df_time**), using the some basic data slicing and iterations steps:

```
23  df_time.head()
```

	year	n_good_ratings	n_other_ratings	n_5star_ratings	n_movies
0	1995	180	74	4	258
1	1994	142	90	5	237
2	1996	180	90	6	276
3	1976	32	7	5	44
4	1992	92	72	3	167

Then we plot the rating counts with respect to time using the plotting function **fn_plot_singlex_multiy** as shown below. This plotting function is very convenient to use while plotting multiple line plots which have the same x-axis. **Multi_y** refers to multiple components (random variables) along the y-axis and **single_x** refers to the common random variable along the x-axis.

```
df = df_time[(df_time.year >= 1960) & (df_time.year < 2000)].sort_values('year')

# X_data & Y_datas:
singlex = df.year
y1 = df.n_movies
y2 = df.n_good_ratings
y3 = df.n_other_ratings
y4 = df.n_5star_ratings
multiy = [y1,y2,y3,y4]

# X_label & Y_labels:
single_x_label = 'YEAR'
l1 = 'n_movies'
l2 = 'n_ratings (3 to 4.5 stars)'
l3 = 'n_ratings( < 3 stars)'
l4 = 'n_ratings = 5 stars'
multiy_labels = [l1, l2, l3, l4]

fn_plot_singlex_multiy(singlex, multiy, single_x_label, multiy_labels)
```



4. VISUALIZING DISTRIBUTION OF MOVIE GENRES:

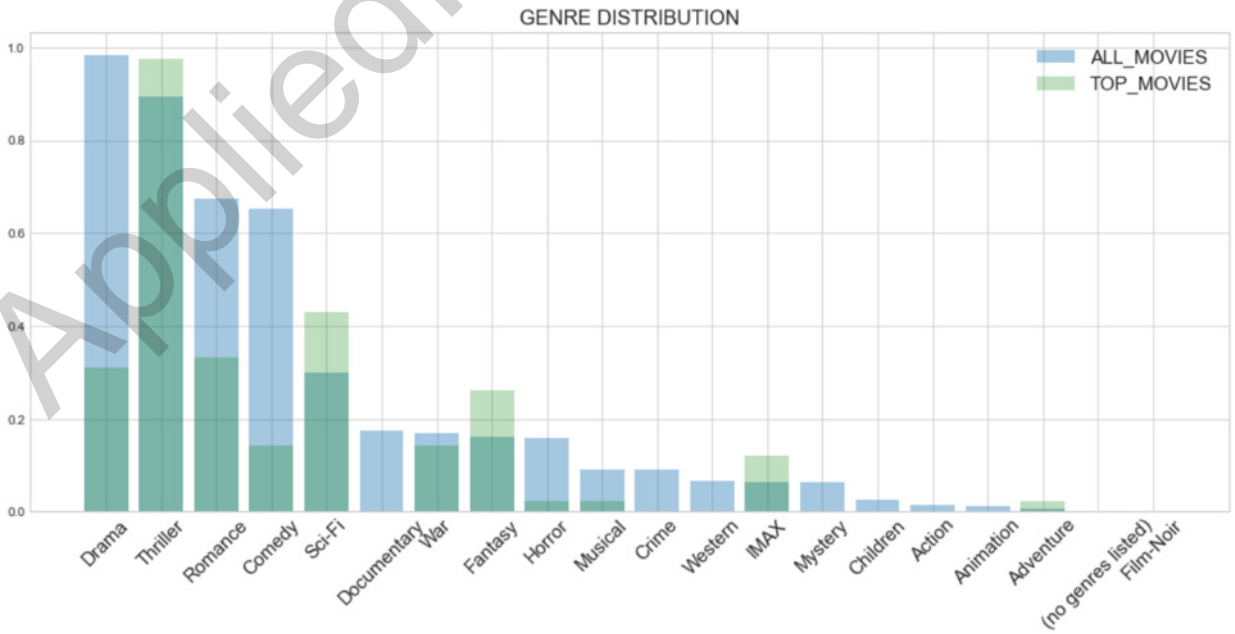
Using the scatter plot created earlier depicting the avg_rating of a movie and the number of times its been watched, we identify the top movies as shown below:

```
df_top_movies = df_movies_meta[df_movies_meta.avg_rating >= 3]
df_top_movies = df_top_movies[df_top_movies.avg_rating <= 4.5]
df_top_movies = df_top_movies[df_top_movies.n_users >100]

df_top_movies.sample(5)
```

	movie_id	title	genre	year	n_users	avg_rating
613	778	Trainspotting	Drama	1996	102	4.039216
1575	2115	Indiana Jones and the Temple of Doom	Fantasy	1984	108	3.638889
334	377	Speed	Thriller	1994	171	3.529240
827	1089	Reservoir Dogs	Thriller	1992	131	4.202290
395	454	Firm, The	Thriller	1993	101	3.534653

We then proceed to implement the intended plot using the code shown below:

```
X1 = df_movies_meta.genre.value_counts().index
X2 = df_top_movies.genre.value_counts().index

genre_dist_all = df_movies_meta.genre.value_counts().values
genre_dist_top = df_top.genre.value_counts().values

def fn_normalize(vals): return (vals - vals.min())/vals.max()
Y1, Y2 = fn_normalize(genre_dist_all), fn_normalize(genre_dist_top)

plt.figure(figsize = (17, 7))

plt.bar(X1, Y1, alpha = 0.4)
plt.bar(X2, Y2, alpha = 0.25, color = 'g')

plt.xticks(rotation = 45, fontsize = 15)
plt.title('GENRE DISTRIBUTION', fontsize=16)

plt.legend(['ALL_MOVIES', 'TOP_MOVIES'], fontsize = 15)
plt.show()
```



The most watched genres when considering the entire dataset are: Drama, thriller, romance and comedy. The most watched genres when considering only the most highly rated movies are: Thriller, Romance, sci-fi and fantasy.

RECOMMENDER SYSTEM FOR MOVIE LENS DATA:

We will be using the surprise python library to perform Collaborative filtering using NMF. The surprise library is specifically designed and optimised for handling datasets that represent user-item ratings. It has a comprehensive suite of functions that aid data splitting, model fitting, hyperparameter tuning, cross validation and testing. It makes it easy to use by keeping all the process sophistication hidden and automating all the boiler plate code.

The data is ingested by the library as shown below:

```
input_data = df_ratings.drop('timestamp', axis = 1)

reader = Reader(rating_scale=(0.5, 5))
data = Dataset.load_from_df(input_data, reader)

data
```

```
<surprise.dataset.DatasetAutoFolds at 0x217de2b5c10>
```

The Reader class shown above, basically helps incorporate the "**rating_scale**" information into the surprise dataset format and the '**load_from_df**' function is used when ratings data is in the form of a dataframe.

The fundamental limitation of collaborative filtering is the "**Cold start problem**". This means that the user and the item of the user-item pair being predicted upon must be present in the training set. It cannot predict if either user or the item in the user-item pair is not present in the training set. The collaborative filtering algorithm can predict only for as yet unseen user-item pairs, as long both the user and the item is present in the training set.

In the view of the above limitation, the train-test split has to be performed carefully such that, all users and items in the testset are present in trainset as combinations with some other items and users respectively.

The surprise library does this for us internally as shown below:

```
tr_set, ts_set = train_test_split(data, test_size=.25)

tr_set, ts_set[:5]
```

```
(<surprise.trainset.Trainset at 0x253deedb490>,
 [(357, 33154, 5.0),
  (474, 7700, 4.0),
  (217, 2152, 2.0),
  (460, 48774, 5.0),
  (243, 589, 4.0)])
```

Note that the train set is converted into a format that is optimized for collaborative filtering, where as the testset is available as a list of tuples that contain user-item-rating information.

The performance metrics used to test collaborative filtering recommender models are the same as those used for regression, ie: RMSE and MAE.

The code for implementing gridsearch with k fold cross validation, on the training set using surprise, is as shown below. In the code below we tune for: number of epochs, 'reg_pu" - which is the regularization term applied to the user matrix and 'reg_gi' - which is the regularization term applied to the item matrix.

```
param_grid = {'n_epochs': [10, 25, 50, 100],
              'reg_pu': [0.08, 0.09, 0.1, 0.15],
              'reg_qi': [0.08, 0.09, 0.1, 0.15]}

n_cv_splits = 3
grid_search_models = GridSearchCV(NMF, param_grid, measures=['rmse', 'mae'], cv=n_cv_splits)

%time grid_search_models.fit(data)
```

```
Wall time: 30min 38s
```

The measures = ['rmse', 'mae'] keyword and argument makes surprise optimize using two regression losses individually, thus yielding the following best prediction models from those two categories.

```
grid_search_models.best_params
```

```
{'rmse': {'n_epochs': 100, 'reg_pu': 0.15, 'reg_qi': 0.15},
 'mae': {'n_epochs': 10, 'reg_pu': 0.15, 'reg_qi': 0.09}}
```

We check the train set performance for two best models from each category as follows:

```
grid_search_models.best_score['rmse'], grid_search_models.best_score['mae']
```

```
(0.9099499830949354, 0.7016054374487055)
```

We test the models as shown below:

A. MAE:

```
testset = ts_set
model_ = grid_search_models.best_estimator['mae'].fit(tr_set)


fn_performance_recommender(testset, model_)
```

	Performance
rmse	0.918515
mae	0.700402
mape	0.309484

B. RMSE

```
testset = ts_set
model_ = grid_search_models.best_estimator['rmse'].fit(tr_set)


fn_performance_recommender(testset, model_)
```

	Performance
rmse	0.905830
mae	0.705178
mape	0.287746

As can be seen from the performances on the testset, the models generalize well to the test set.