KURUNJI VENKATRAMANA GOWDA POLYTECHNIC SULLIA-574327

5TH SEMESTER

AI/ML WEEK-8

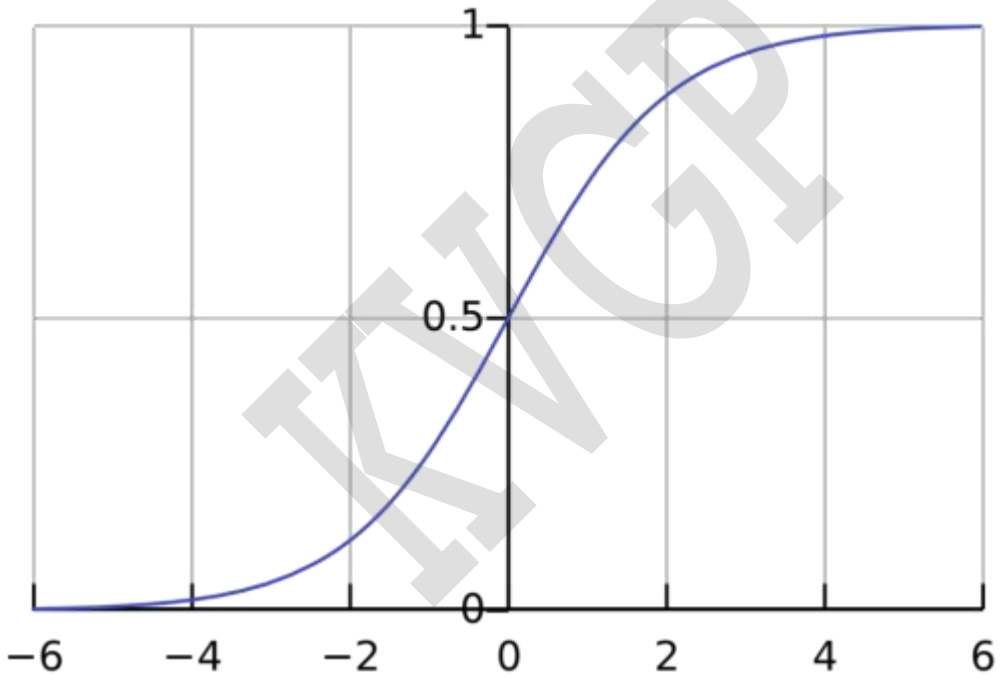
Logistic Regression:

Overview: what is logistic regression

Logistic regression is a machine learning method used in the classification problem when you need to distinguish one class from another. The simplest case is a binary classification. This is like a question that we can answer with either "yes" or "no." We only have two classes: a positive class and negative class. Usually, a positive class points to the presence of some entity while negative class points to the absence of it.

In this case, we need to predict a single value - the probability that entity is present. To do so, it will be good for us to have a function that maps any real value to value in the interval between 0 and 1.

Let's look at this function plot.



It shows a pretty decent mapping between R and the (0, 1) interval. It suits our requirements.

This is the so-called sigmoid function and it is defined this way:

$$f(x) = \frac{1}{1 + e^{-x}}$$

Most far from 0 values of x are mapped close to 0 or close to 1 values of y.

Values close to 0 of *x* will be a good approximation of probability in our algorithm. Then we can choose a threshold value and transform probability to 0 or 1 prediction.

Sigmoid is an activation function for logistic regression. Now let's define the cost function for our optimization algorithm.

The first thing that comes into mind when we think about cost function is a classic square error function.

$$J(\Theta) = \frac{1}{2m}\sum_{i=1}^{m}(h_\Theta(x^{(i)}) - y^{(i)})^2$$

Where *m* - number of examples, $x^{(i)}$ - feature vector for *i*-th example, $y^{(i)}$ - actual value for *i*-th example, θ - parameters vector.

If we have a linear activation function $h_\theta(x)$ then it's okay. But with our new sigmoid function, we have no positive second derivative for square error. It means that it is not convex. We don't want to stuck in local optima, thus we define a new cost function:

$$J(\Theta) = -\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)} * \log(h_\Theta(x^{(i)})) + (1 - y^{(i)}) * \log(1 - h_\Theta(x^{(i)}))\right]$$

This is called a cross-entropy cost. If you look carefully, you may notice that when a prediction is close to actual value then cost will be close to zero for both 0 and 1 actual values.

Let's suppose we have features $x_1, x_2, ..., x_n$, and y value for every entity.

Then we have $n+1$-dimensioned θ parameters vector, such that:

$$h_\Theta(x) = sigmoid(\Theta_0 + \Theta_1 x_1 + ... + \Theta_n x_n)$$

Representation of Logistic regression

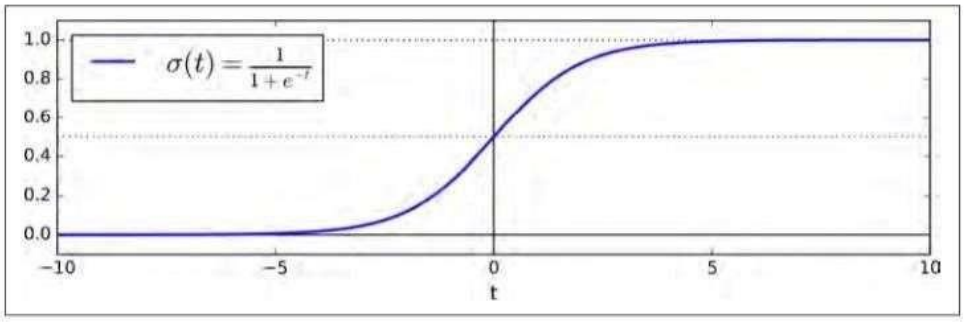
Just like a lInear regression model, a logistic regression model also computes a weighted sum of input features(and adds a bias term to it). However, unlike linear regression, it calculates the logistic of the results so that the output is always between o and
1.

Logistic Regression model estimated probability (vectorized form)

$$\hat{p} = h_\theta(\mathbf{x}) = \sigma\left(\theta^T \cdot \mathbf{x}\right)$$

The logisitc also called as logit is denoted by σ is a sigmoid function and it outputs a number between 0 and 1.

$$\sigma(t) = \frac{1}{1 + \exp(-t)}$$



- Positive values are predictive of class 1
- Negative values are predictive of class 0

The output of a Logistic regression model is a probability. We can select a threshold value. If the probability is greater than this threshold value, the event is predicted to happen otherwise it is predicted not to happen.

Once the mpdel estimates the probabilities(p hat), it can then easily make the predictions as follows:

$$\hat{y} = \begin{cases} 0 \text{ if } \hat{p} < 0.5, \\ 1 \text{ if } \hat{p} \geq 0.5. \end{cases}$$

And we optimize θ with gradient descent and cross-entropy cost.
That's it!

Types:

There are three main types of logistic regression:

1. **Binary**

   Binary logistic regression (LR) is **a regression model where the target variable is binary**, that is, it can take only two values, 0 or 1. It is the most utilized regression model in readmission prediction, given that the output is modelled as readmitted (1) or not readmitted (0).

   For binary logistic regression, the odds of success are: **$\pi 1-\pi=\exp(X\beta)$**. f_0By plugging this into the formula for θ above and setting X(1) equal to X(2) except in one position (i.e., only one predictor differs by one unit), we can determine the relationship between that predictor and the response.

2. **Multinomial**

   Multinomial logistic regression is **used to predict categorical placement in or the probability of category membership on a dependent variable based on multiple independent variables**. The independent variables can be either dichotomous (i.e., binary) or continuous (i.e., interval or ratio in scale).

   Also, it gives a good insight on what the multinomial logistic regression is: a set of J−1 independent logistic regressions for the probability of Y=j versus the probability of the reference Y=J. **Y = J . pj(x)=eβ0j+β1jX1+..+βpjXppJ(x)**.

3. **ordinal**.

   Ordinal logistic regression is **a statistical analysis method that can be used to model the relationship between an ordinal response variable and one or more explanatory variables**. An ordinal variable is a categorical variable for which there is a clear ordering of the category levels.

   Ordinal Logistic Regression Model **l o g i t (P (Y ≤ j)) = β j 0 + β 1 x 1 + … + β p x p** .

How does logistic Regression works?

Assumptions of logistic regression

- **Response variable is binary**

It basically the whole point of logistic regression. It assumes that the response variable or dependent variable can give only two variables.

- Yes/No
- True/False
- Disable/Enable
- In/Out

The simple way to measure this assumptions to find out how many unique outcomes the response variable can possibly give.

- **Observations are independent**

Logistic regression assumes the observations to be independent of each other and independent of repetitive measurement. Any individual should not be measured more than once and neither should it be taken in for the model.

A way to check this assumptions is by maintaining an order for the observations. You need to make sure the observations are done at random without any biases, or else the assumption get violated.

- **Explanatory variable shave no multicollinearity**

Multicollinearity in explanatory variables occurs when two or more than two of them do not provide unique information to the model. In this case, the explanatory are correlated to each other and provide similar information. In case of high correlativity between variables, they will create discrepancies while fitting in the interpreting regression model.

Let's say you want to observe the weight of babies, the observations for the following would be:

- Weight of the baby
- Baby's clothes' weight
- Baby's diet

Here, the weight of the baby and its clothes are the variables that give out more or less the same data, further taking up the space in the model.

The best way to look out for multicollinearity is to use VIF (variance inflation factor). It is a way to measure the correlation and its strength between the explanatory variables.

- **No extreme outliers**

Logistic regression assumes that there are no extreme outliers or any external observations that influence the data that goes into the model.

Cook's distance is an effective way to rule out the outliers and external observations from a dataset. You can choose to eradicate those from the data or decide to replace them with a mean or median. You can also let the outliers be, but remember to report those in the regression results.

- **The explanatory variables and the Logit of response variable have a linear relationship between them.**

The Logit is stated as:

Logit(p) = log(p / (1-p))

Where p is the probability of an outcome to be positive.

The logistic regression assumes that this Logit of the response variable and the explanatory variables are linearly related.

Box-Tidwell test is used to see if this assumption stands true in your dataset for the regression model.

- **Sufficient sample size**

The logistic regression assumes that the sample size from which the observations are drawn is large enough to give reliable conclusions for the regression model.

There is a rule of thumb to put this assumption in place. You need to have at least 10 cases where the outcome is not very frequent, for each explanatory variable. Let's say you have 5 explanatory variables and you are expecting the probability of the least frequent outcome turns out to be 0.30, the model demands the sample size of at least (10*5)/0.30 = 166.

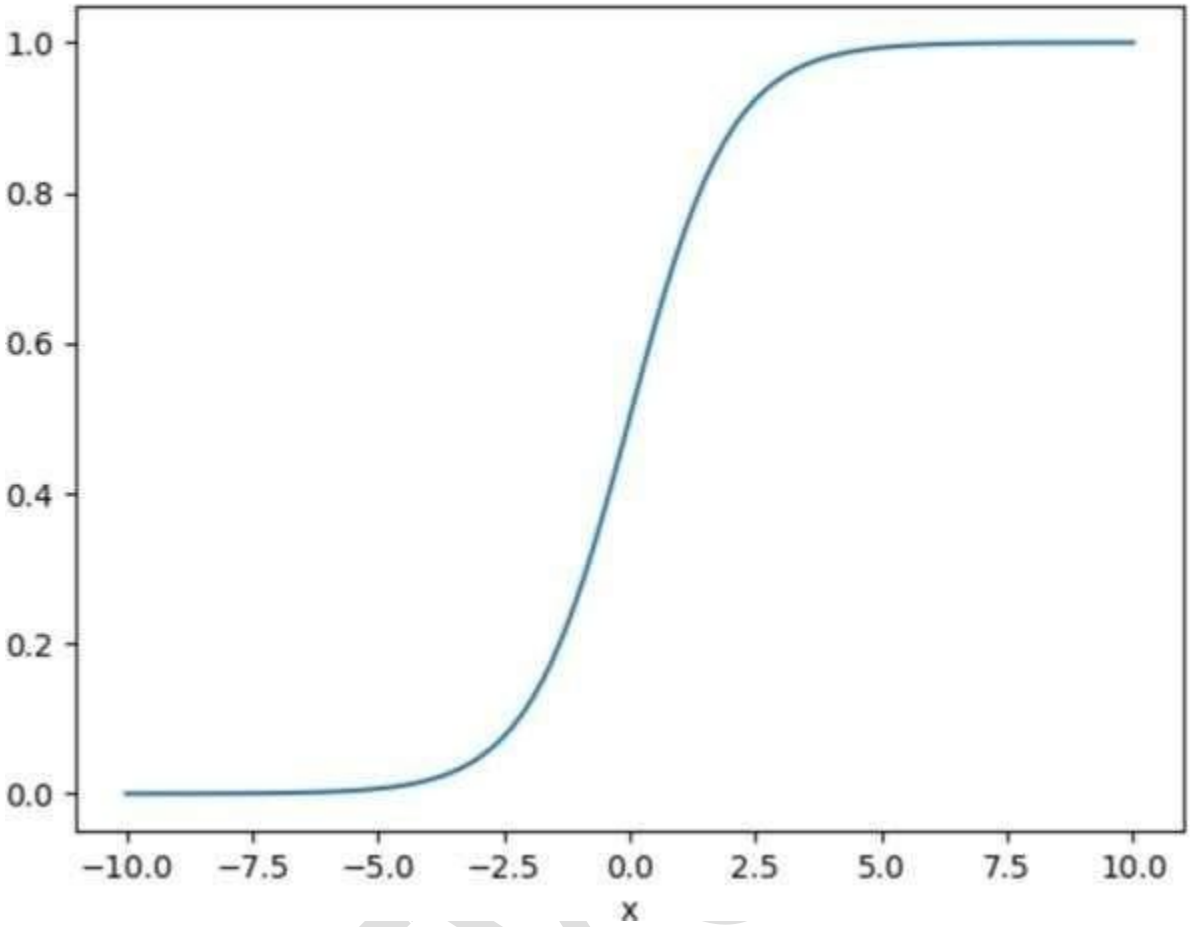
- **Both logistic regression and linear regression have common assumptions:**
- A linear relationship between the explanatory variables and the response variable.
- Normally distributed residuals.
- Homoscedasticity between the residuals.

What is a sigmoid function?

The logistic function in linear regression is a type of sigmoid, a class of functions with the same specific properties.

Sigmoid is a mathematical function that takes any real number and maps it to a probability between 1 and 0.

The formula of the sigmoid function is:



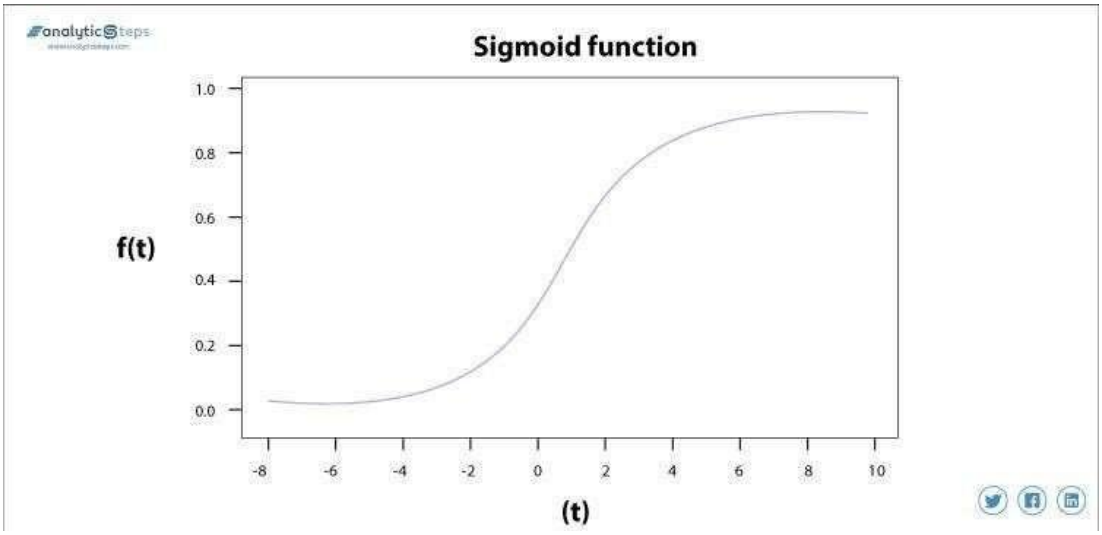
The sigmoid function forms an *S* shaped graph, which means as x*x* approaches infinity, the probability becomes 1, and as x*x* approaches negative infinity, the probability becomes 0. The model sets a threshold that decides what range of probability is mapped to which binary variable.

Suppose we have two possible outcomes, *true* and *false*, and have set the threshold as *0.5*. A probability less than *0.5* would be mapped to the outcome *false*, and a probability greater than or equal to *0.5* would be mapped to the outcome *true*.

It is a mathematical function having a characteristic that can take any real value and map it to between 0 to 1 shaped like the letter “S”. The sigmoid function also called a logistic function.

$$Y = 1 / 1+e^{-z}$$

Sigmoid function



So, if the value of z goes to positive infinity then the predicted value of y will become 1 and if it goes to negative infinity then the predicted value of y will become 0. And if the outcome of the sigmoid function is more than 0.5 then we classify that label as class 1 or positive class and if it is less than 0.5 then we can classify it to negative class or label as class 0.

Why do we use the Sigmoid Function?

Sigmoid Function acts as an activation function in machine learning which is used to add non-linearity in a machine learning model, in simple words it decides which value to pass as output and what not to pass, there are mainly 7 types of Activation Functions which are used in machine learning and deep learning.

Applications of logistic regression:

Applications. Logistic regression is used in various fields, including **machine learning, most medical fields, and social sciences**. For example, the Trauma and Injury Severity Score (TRISS), which is widely used to predict mortality in injured patients, was originally developed by Boyd et al. using logistic regression.

5 real-world cases where logistic regression was effectively used

Credit scoring

ID Finance is a financial company that makes predictive models for credit scoring. They need their models to be easily interpretable. They can be asked by a regulator about a certain decision at any moment.

Data preprocessing for credit scoring modeling includes such a step like reducing correlated variables. It's difficult if you have more than 15 variables in your model. For logistic regression, it is easy to find out which variables affect the final result of the predictions more and which ones less. It is also possible to find the optimal number of features and eliminate redundant variables with methods like recursive feature elimination.

At the final step, they can export prediction results to an Excel file, and analytic even without technical skills can get insights from this data.

At some point, ID finance refused the use of third-party statistical applications and rewrote their algorithms for building models in Python. This has led to a significant increase in the speed of model development. But they did not abandon logistic regression in favor of more complex algorithms. Logistic regression is widely used in credit scoring and it shows remarkable results.

Medicine

Medical information is gathered in such a way that when a research group studies a biological molecule and its properties, they publish a paper about it. Thus, there is a huge amount of medical data about various compounds, but they are not combined into a single database.

Miroculus is a company that develops express blood test kits. Its goal is to identify diseases that are affected by genes, such as oncology diseases. The company entered into an agreement with Microsoft to develop an algorithm to identify the relationship between certain micro-RNA and genes.

The developers used a database of scientific articles and applied text analysis methods to obtain feature vectors. The text was split into the sentences, the entities were extracted, labeled data generated from known relations, and after several other text transformation methods, each sentence was converted into a 200-dimensional vector.

After converting the text and extracting the distinguishing features, a classification was made for the presence of a link between microRNA and a certain gene. Algorithms such as logistic regression, support vector machine,

and random forest were considered as models. Logistic regression was selected because it demonstrated the best results in speed and accuracy.

Logistic regression is well suited for this data type when we need to predict a binary answer. Is there a connection between the elements or not? Thanks to this algorithm, the accuracy of a quick blood test have been increased.

Text editing

As we talked about texts, it is worth mentioning that logistic regression is a popular choice in many natural language processing tasks. First, the text preprocessing is performed, then features are extracted, and finally, logistic regression is used to make some claim about a text fragment. Toxic speech detection, topic classification for questions to support, and email sorting are examples where logistic regression shows good results. Other popular algorithms for making a decision in these fields are support vector machines and random forest.

Let's look at the less popular NLP task - text transformation or digitalization. One company has faced this problem: they had a lot of PDF text files and texts extracted from scans with the OCR system. Such files had a fixed structure with line break by the characters of the end of the paragraph, and with hyphens. They needed to transform this data into usable text with grammatical and semantic correct formatting.

The developer manually marked out three large documents, adding special characters to the beginning of the line indicating whether it should be glued to the previous line. As features were chosen: the length of the current and previous lines in characters, the average length of several lines around, whether the last character of the previous line is a letter or a digit, punctuation mark on which the previous line ends, and some other properties. All string and boolean features were transformed into numerical. Then logistic regression was trained.

It showed a few errors and these were mainly the same errors that humans can make in such a situation. There were very few easy human-readable errors. Logistic regression showed excellent results in this task, and a lot of texts were automatically transformed using this method.

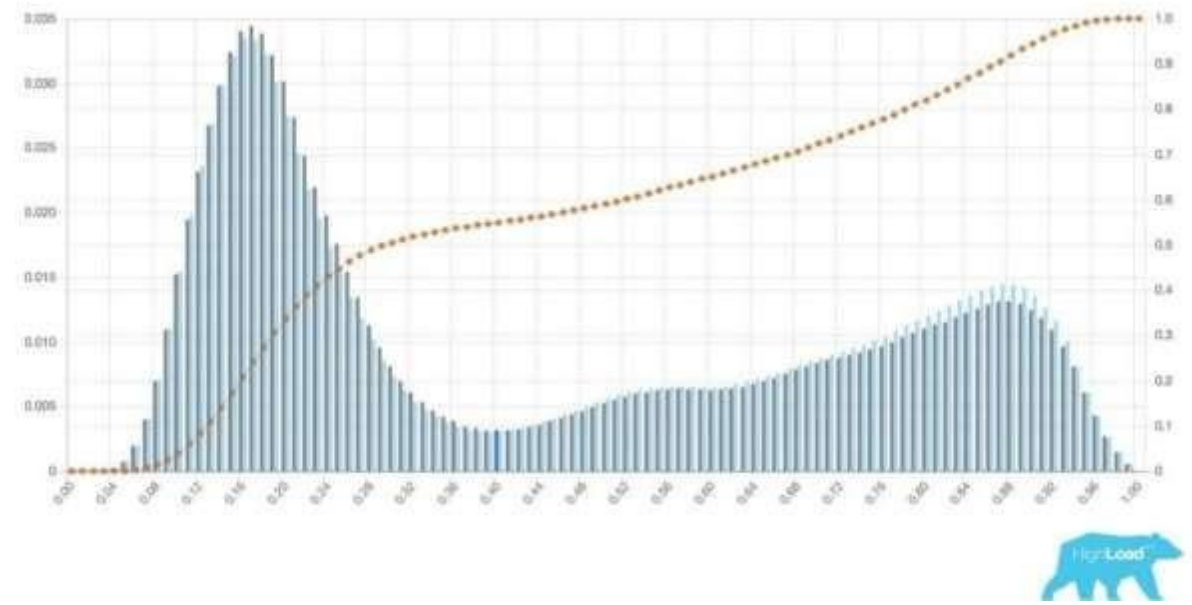
Hotel Booking

Booking.com has a lot of machine learning methods literally everywhere on the site. They try to predict users' intentions and recognize entities. Where will you go, where do you prefer to stop, what are you planning to do? Some

predictions are made even if the user didn't type anything in the search line yet. But how did they start to do this? No one can build a huge and complex system with various machine learning algorithms from scratch. They have accumulated some statistics and created some simple models as the first steps.

Most of the features at such services like booking.com are rather categorical than numerical. Sometimes it becomes necessary to predict an event without specific data about the user. For example, all the data they have is where the user is from and where she wants to go. Logistic regression is ideal for such needs.

Here is a histogram of logistic regression trying to predict either user will change a journey date or not. It was presented at HighLoad++ Siberia conference in 2018.



Logistic regression could well separate two classes of users. Based on this data, the company then can decide if it will change an interface for one class of users.

You probably saw this functionality if you have used Booking. Now you know there is logistic regression somewhere behind this application.

Gaming

Speed is one of the advantages of logistic regression, and it is extremely useful in the gaming industry. Speed is very important in a game. Very popular today are the games where you can use in-game purchases to improve the gaming qualities of your character, or for fancy appearance and communication with other players. In-game purchases are a good place to introduce a recommendation system.

Tencent is the world's largest gaming company. It uses such systems to suggest gamers' equipment which they would like to buy. Their algorithm analyzes a very large amount of data about user behavior and gives suggestions about equipment a particular user may want to acquire on the run. This algorithm is logistic regression.
There are three types of recommendation systems. The collaborative system predicts what the user would like to buy based on ratings from users with similar preferences in previous purchases, and other activity. A content-based algorithm makes its decision based on properties specified in the item description and what the user indicated as interests in her profile. The third type is the hybrid and it is a combination of two previous types.

Both the description and the preferences of other users can be used as features in logistic regression. You only need to transform them into a similar format and normalize. Logistic regression will work fast and show good results.

Logistic Regression Model in Python:

Use the above
link to get model:
https://www.kaggle.com/code/stieranka/logistic-regression-withpython/notebook

Building A Logistic Regression in Python, Step by

Logistic Regression is a Machine Learning classification algorithm that is used to predict the probability of a categorical dependent variable. In logistic regression, the dependent variable is a binary variable that contains data coded as 1 (yes, success, etc.) or 0 (no, failure, etc.). In other words, the logistic regression model predicts P(Y=1) as a function of X.

Logistic Regression Assumptions

- Binary logistic regression requires the dependent variable to be binary.

- For a binary regression, the factor level 1 of the dependent variable should represent the desired outcome.
- Only the meaningful variables should be included.
- The independent variables should be independent of each other.

That is, the model should have little or no multicollinearity.

- The independent variables are linearly related to the log odds.
- Logistic regression requires quite large sample sizes.

Keeping the above assumptions in mind, let's look at our dataset.

Data

The dataset comes from the UCI Machine Learning repository, and it is related to direct marketing campaigns (phone calls) of a Portuguese banking institution. The classification goal is to predict whether the client will subscribe (1/0) to a term deposit (variable y). The dataset can be downloaded from here.

```
import pandas as pd import numpy as
np
from sklearn import preprocessing import
matplotlib.pyplot as plt plt.rc("font", size=14)
from sklearn.linear_model import
LogisticRegression from sklearn.model_selection import
train_test_split import seaborn as sns sns.set(style="white")
sns.set(style="whitegrid", color_codes=True)
```

The dataset provides the bank customers' information. It includes 41,188 records and 21 fields.

```
In [39]: data = pd.read_csv('bank.csv', header=0)
         data = data.dropna()
         print(data.shape)
         print(list(data.columns))

         (41188, 21)
         ['age', 'job', 'marital', 'education', 'default', 'housing', 'loan', 'contact', 'month', 'day_of_week', 'duration', 'campaign',
         'pdays', 'previous', 'poutcome', 'emp_var_rate', 'cons_price_idx', 'cons_conf_idx', 'euribor3m', 'nr_employed', 'y']

In [37]: data.head()

Out[37]:
```

	age	job	marital	education	default	housing	loan	contact	month	day_of_week	...	campaign	pdays	previous	poutcome	emp_var_rate
0	44	blue-collar	married	basic.4y	unknown	yes	no	cellular	aug	thu	...	1	999	0	nonexistent	1.4
1	53	technician	married	unknown	no	no	no	cellular	nov	fri	...	1	999	0	nonexistent	-0.1
2	28	management	single	university.degree	no	yes	no	cellular	jun	thu	...	3	6	2	success	-1.7
3	39	services	married	high.school	no	no	no	cellular	apr	fri	...	2	999	0	nonexistent	-1.8
4	55	retired	married	basic.4y	no	yes	no	cellular	aug	fri	...	1	3	1	success	-2.9

5 rows × 21 columns

Figure 1

Input variables

1. age (numeric)

2. job : type of job (categorical: “admin”, “bluecollar”, “entrepreneur”,
   “housemaid”, “management”, “retired”, “selfemployed”,
   “services”, “student”, “technician”,
   “unemployed”, “unknown”)

3. marital : marital status (categorical: “divorced”, “married”, “single”, “unknown”)
4. education (categorical: “basic.4y”, “basic.6y”, “basic.9y”,
   “high.school”, “illiterate”, “professional.course”,
   “university.degree”, “unknown”)

5. default: has credit in default? (categorical: “no”, “yes”, “unknown”)
6. housing: has housing loan? (categorical: “no”, “yes”, “unknown”)

7. loan: has personal loan? (categorical: “no”, “yes”, “unknown”)

8. contact: contact communication type (categorical: "cellular", "telephone")

9. month: last contact month of year (categorical: "jan", "feb", "mar", …, "nov", "dec")

10. day_of_week: last contact day of the week (categorical: "mon", "tue", "wed", "thu", "fri")

11. duration: last contact duration, in seconds (numeric). Important note: this attribute highly affects the output target (e.g., if duration=0 then y='no'). The duration is not known before a call is performed, also, after the end of the call, y is obviously known. Thus, this input should only be included for benchmark purposes and should be discarded if the intention is to have a realistic predictive model

12. campaign: number of contacts performed during this campaign and for this client (numeric, includes last contact)

13. pdays: number of days that passed by after the client was last contacted from a previous campaign (numeric; 999 means client was not previously contacted)

14. previous: number of contacts performed before this campaign and for this client (numeric)

15. poutcome: outcome of the previous marketing campaign (categorical: "failure", "nonexistent", "success")

16. emp.var.rate: employment variation rate — (numeric)

17. cons.price.idx: consumer price index — (numeric)

18. cons.conf.idx: consumer confidence index — (numeric)

19. euribor3m: euribor 3 month rate — (numeric)

20. nr.employed: number of employees — (numeric)

Predict variable (desired target):

y — has the client subscribed a term deposit? (binary: "1", means "Yes", "0" means "No")

The education column of the dataset has many categories and we need to reduce the categories for a better modelling. The education column has the following categories:

```
In [4]: data['education'].unique()
Out[4]: array(['basic.4y', 'unknown', 'university.degree', 'high.school',
               'basic.9y', 'professional.course', 'basic.6y', 'illiterate'], dtype=object)
```

Figure 2

Let us group "basic.4y", "basic.9y" and "basic.6y" together and call them "basic".

```
data['education']=np.where(data['education'] =='basic.9y', 'Basic', data['education'])
data['education']=np.where(data['education'] =='basic.6y', 'Basic', data['education'])
data['education']=np.where(data['education'] =='basic.4y', 'Basic', data['education'])
```

After grouping, this is the columns:

```
In [6]: data['education'].unique()
Out[6]: array(['Basic', 'unknown', 'university.degree', 'high.school',
               'professional.course', 'illiterate'], dtype=object)
```

Figure 3

Data exploration

```
In [7]: data['y'].value_counts()

Out[7]: 0    36548
        1     4640
        Name: y, dtype: int64
```

```
In [17]: sns.countplot(x='y',data=data, palette='hls')
         plt.show()
         plt.savefig('count_plot')
```

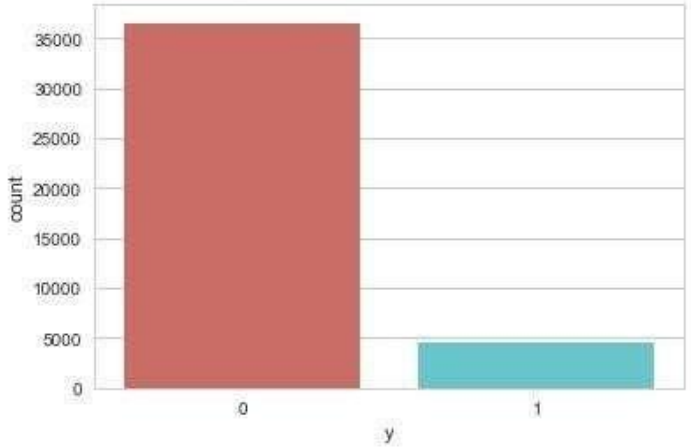


Figure 4

count_no_sub = len(data[data['y']==0]) count_sub = len(data[data['y']==1])

```
pct_of_no_sub = count_no_sub/(count_no_sub+count_sub)
print("percentage of no subscription is", pct_of_no_sub*100)
pct_of_sub = count_sub/(count_no_sub+count_sub)
print("percentage of subscription", pct_of_sub*100)
```

percentage of no subscription is 88.73458288821988 percentage of subscription 11.265417111780131

Our classes are imbalanced, and the ratio of no-subscription to subscription instances is 89:11. Before we go ahead to balance the classes, let's do some more exploration.

```
In [9]: data.groupby('y').mean()
```

Out[9]:

y	age	duration	campaign	pdays	previous	emp_var_rate	cons_price_idx	cons_conf_idx	euribor3m	nr_employed
0	39.911185	220.844807	2.633085	984.113878	0.132374	0.248875	93.603757	-40.593097	3.811491	5176.166600
1	40.913147	553.191164	2.051724	792.035560	0.492672	-1.233448	93.354386	-39.789784	2.123135	5095.115991

Figure 5

Observations:

- The average age of customers who bought the term deposit is higher than that of the customers who didn't.
- The pdays (days since the customer was last contacted) is understandably lower for the customers who bought it. The lower the pdays, the better the memory of the last call and hence the better chances of a sale.
- Surprisingly, campaigns (number of contacts or calls made during the current campaign) are lower for customers who bought the term deposit.

We can calculate categorical means for other categorical variables such as education and marital status to get a more detailed sense of our data.

```
In [10]: data.groupby('job').mean()
```

Out[10]:

job	age	duration	campaign	pdays	previous	emp_var_rate	cons_price_idx	cons_conf_idx	euribor3m	nr_employed	y
admin.	38.187296	254.312128	2.623489	954.319229	0.189023	0.015563	93.534054	-40.245433	3.550274	5164.125350	0.129726
blue-collar	39.555760	264.542360	2.558461	985.160363	0.122542	0.248995	93.656656	-41.375816	3.771996	5175.615150	0.068943
entrepreneur	41.723214	263.267857	2.535714	981.267170	0.138736	0.158723	93.605372	-41.283654	3.791120	5176.313530	0.085165
housemaid	45.500000	250.454717	2.639623	960.579245	0.137736	0.433396	93.676576	-39.495283	4.009645	5179.529623	0.100000
management	42.362859	257.058140	2.476060	962.647059	0.185021	-0.012688	93.522755	-40.489466	3.611316	5166.650513	0.112175
retired	62.027326	273.712209	2.476744	897.936047	0.327326	-0.698314	93.430786	-38.573081	2.770066	5122.262151	0.252326
self-employed	39.949331	264.142153	2.660802	976.621393	0.143561	0.094159	93.559982	-40.488107	3.689376	5170.674384	0.104856
services	37.926430	258.398085	2.587805	979.974049	0.154951	0.175359	93.634659	-41.290048	3.699187	5171.600126	0.081381
student	25.894857	283.683429	2.104000	840.217143	0.524571	-1.408000	93.331613	-40.187543	1.884224	5085.939088	0.314286
technician	38.507638	250.232241	2.577339	964.408127	0.153789	0.274566	93.561471	-39.927569	3.820401	5175.648391	0.108260
unemployed	39.733728	249.451677	2.564103	935.316568	0.199211	-0.111736	93.563781	-40.007594	3.466583	5157.156509	0.142012
unknown	45.563636	239.675758	2.648485	938.727273	0.154545	0.357879	93.718942	-38.797879	3.949033	5172.931818	0.112121

Figure 6

```
In [11]: data.groupby('marital').mean()
```

Out[11]:

marital	age	duration	campaign	pdays	previous	emp_var_rate	cons_price_idx	cons_conf_idx	euribor3m	nr_employed	y
divorced	44.899393	253.790330	2.61340	968.639853	0.168690	0.163985	93.606563	-40.707069	3.715603	5170.878643	0.103209
married	42.307165	257.438623	2.57281	967.247673	0.155608	0.183625	93.597367	-40.270659	3.745832	5171.848772	0.101573
single	33.158714	261.524378	2.53380	949.909578	0.211359	-0.167989	93.517300	-40.918698	3.317447	5155.199265	0.140041
unknown	40.275000	312.725000	3.18750	937.100000	0.275000	-0.221250	93.471250	-40.820000	3.313038	5157.393750	0.150000

```
In [12]: data.groupby('education').mean()
```

Out[12]:

education	age	duration	campaign	pdays	previous	emp_var_rate	cons_price_idx	cons_conf_idx	euribor3m	nr_employed	y
Basic	42.163910	263.043874	2.559498	974.877967	0.141053	0.191329	93.639933	-40.927595	3.729654	5172.014113	0.087029
high.school	37.998213	260.886810	2.568576	964.358382	0.185917	0.032937	93.584857	-40.940641	3.556157	5164.994735	0.108355
illiterate	48.500000	276.777778	2.277778	943.833333	0.111111	-0.133333	93.317333	-39.950000	3.516556	5171.777778	0.222222
professional.course	40.080107	252.533855	2.586115	960.765974	0.163075	0.173012	93.569864	-40.124108	3.710457	5170.155979	0.113485
university.degree	38.879191	253.223373	2.563527	951.887892	0.192390	-0.026090	93.493466	-39.975805	3.529663	5163.226298	0.137245
unknown	43.481225	262.390526	2.596187	942.830734	0.226459	0.059099	93.658615	-39.877816	3.571098	5169.549509	0.145003

Figure 7

Visualizations

```
%matplotlib inline
```

```
pd.crosstab(data.job,data.y).plot(kind='bar') plt.title('Purchase Frequency for Job Title') plt.xlabel('Job')
plt.ylabel('Frequency of Purchase') plt.savefig('purchase_fre_job')
```

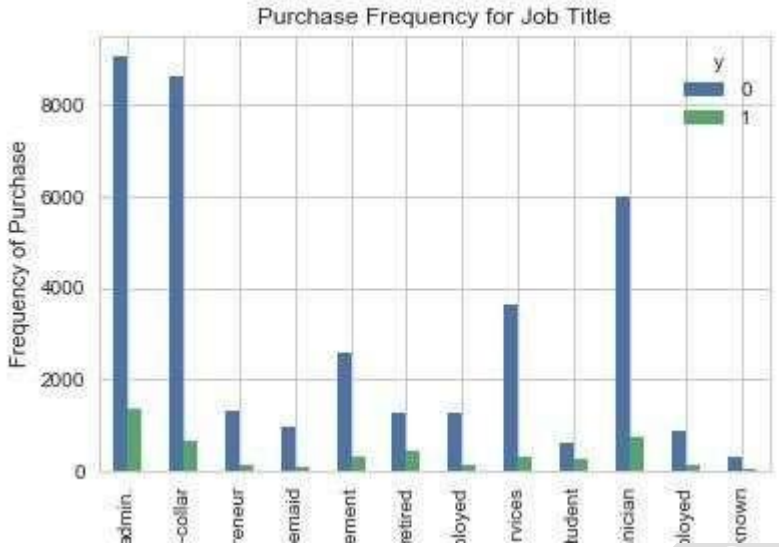


Figure 8

The frequency of purchase of the deposit depends a great deal on the job title.

Thus, the job title can be a good predictor of the outcome variable.

```
table=pd.crosstab(data.marital,data.y)
table.div(table.sum(1).astype(float), axis=0).plot(kind='bar', stacked=True)
plt.title('Stacked Bar Chart of Marital Status vs Purchase') plt.xlabel('Marital Status')
plt.ylabel('Proportion of Customers') plt.savefig('mariral_vs_pur_stack')
```

Figure 9

Education seems a good predictor of the outcome variable.

```
pd.crosstab(data.day_of_week,data.y).p lot(kind='bar')
```

The marital status does not seem a strong predictor for the outcome variable.

```
table=pd.crosstab(data.education,data.y)
table.div(table.sum(1).astype(float), axis=0).plot(kind='bar',
stacked=True)
plt.title('Stacked Bar Chart of Education vs Purchase')
plt.xlabel('Education')
plt.ylabel('Proportion of Customers')
plt.savefig('edu_vs_pur_stack')
```

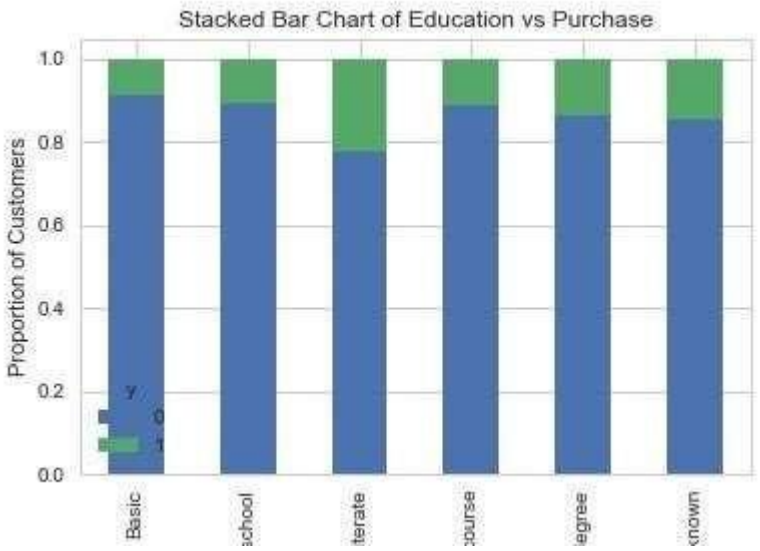


Figure 10

```
plt.title('Purchase Frequency for Day of Week') plt.xlabel('Day
of Week') plt.ylabel('Frequency of Purchase')
plt.savefig('pur_dayofweek_bar')
```

Figure 11

Day of week may not be a good predictor of the outcome.

```
pd.crosstab(data.month,data.y).plot(kind='bar') plt.title('Purchase Frequency for Month') plt.xlabel('Month')
plt.ylabel('Frequency of Purchase') plt.savefig('pur_fre_month_bar')
```

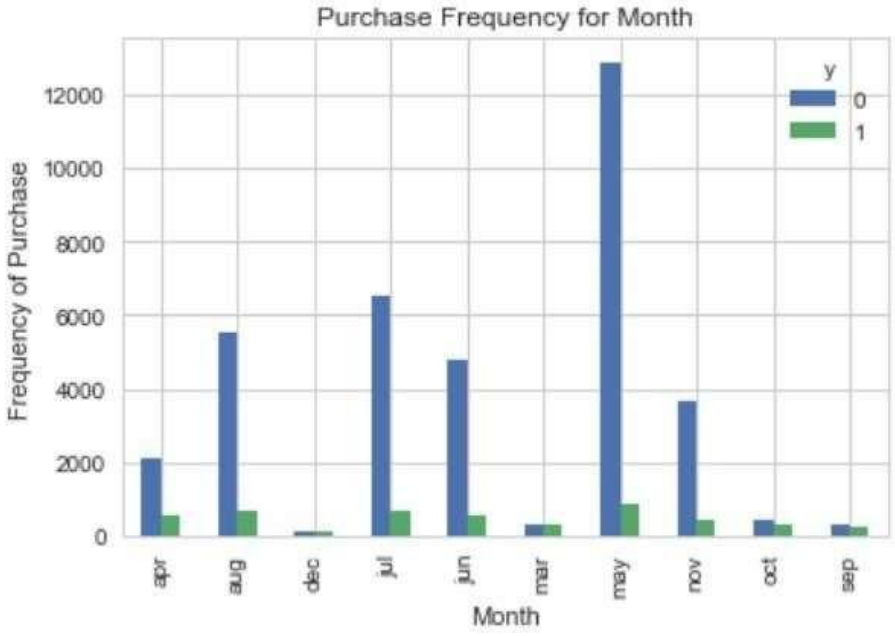


Figure 12

Month might be a good predictor of the outcome variable.

```
data.age.hist()  plt.title('Histogram of Age')
plt.xlabel('Age') plt.ylabel('Frequency')
plt.savefig('hist_age')
```

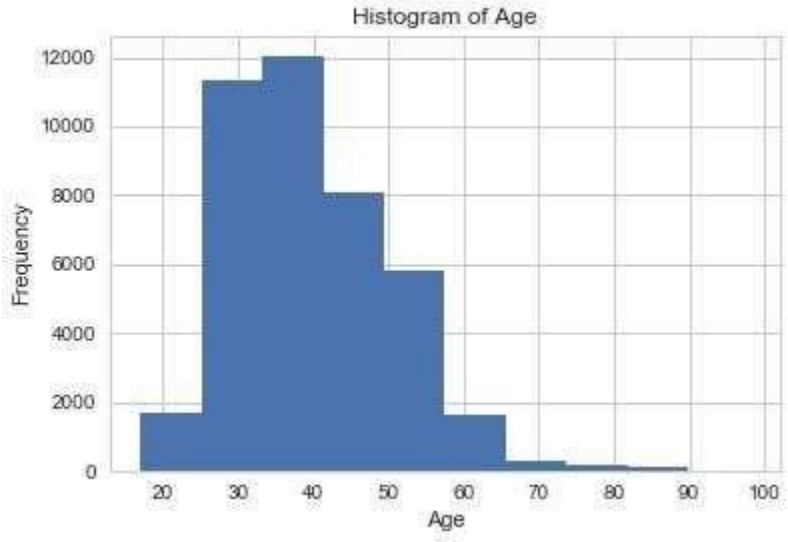


Figure 13

Most of the customers of the bank in this dataset are in the age range of 30–40.

```
pd.crosstab(data.poutcome,data.y).p lot(kind='bar')
plt.title('Purchase Frequency for Poutcome')
```

```
plt.xlabel('Poutcome') plt.ylabel('Frequency of Purchase')
plt.savefig('pur_fre_pout_bar')
```

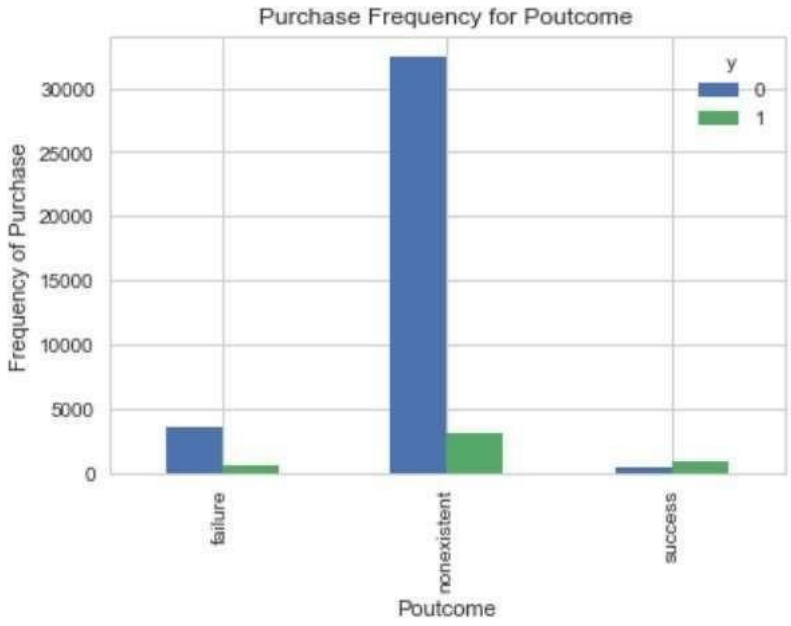


Figure 14

Poutcome seems to be a good predictor of the outcome variable.

Create dummy variables

That is variables with only two values, zero and one.

```
cat_vars=['job','marital','education','default','ho using','loan','c
ontact','month','day_of_week','poutcome'] for var in cat_vars:
   cat_list='var'+'_'+var                          cat_list      =
pd.get_dummies(data[var],                           prefix=var)
data1=data.join(cat_list)
    data=data1cat_vars=['job','marital','education','de fault','housing'
,'loan','contact','month','day_of_we ek','poutcome']
data_vars=data.columns.values.tolist () to_keep=[i for
i in data_vars if i not in cat_vars]
```

Our final data columns will be:

```
data_final=data[to_keep]
data_final.columns.values
```

```
array(['age', 'duration', 'campaign', 'pdays', 'previous', 'emp_var_rate',
       'cons_price_idx', 'cons_conf_idx', 'euribor3m', 'nr_employed', 'y',
       'job_admin.', 'job_blue-collar', 'job_entrepreneur',
       'job_housemaid', 'job_management', 'job_retired',
       'job_self-employed', 'job_services', 'job_student',
       'job_technician', 'job_unemployed', 'job_unknown',
       'marital_divorced', 'marital_married', 'marital_single',
       'marital_unknown', 'education_Basic', 'education_high.school',
       'education_illiterate', 'education_professional.course',
       'education_university.degree', 'education_unknown', 'default_no',
       'default_unknown', 'default_yes', 'housing_no', 'housing_unknown',
       'housing_yes', 'loan_no', 'loan_unknown', 'loan_yes',
       'contact_cellular', 'contact_telephone', 'month_apr', 'month_aug',
       'month_dec', 'month_jul', 'month_jun', 'month_mar', 'month_may',
       'month_nov', 'month_oct', 'month_sep', 'day_of_week_fri',
       'day_of_week_mon', 'day_of_week_thu', 'day_of_week_tue',
       'day_of_week_wed', 'poutcome_failure', 'poutcome_nonexistent',
       'poutcome_success'], dtype=object)
```

Figure 15

Over-sampling using SMOTE

With our training data created, I'll up-sample the nosubscription using the SMOTE algorithm(Synthetic Minority Oversampling Technique). At a high level, SMOTE:

1. Works by creating synthetic samples from the minor class (nosubscription) instead of creating copies.

2. Randomly choosing one of the k-nearest-neighbors and using it to create a similar, but randomly tweaked, new observations.

We are going to implement SMOTE in Python.

```
X = data_final.loc[:, data_final.columns != 'y'] y = data_final.loc[:,
data_final.columns == 'y']from
imblearn.over_sampling import SMOTEos = SMOTE(random_state=0) X_train, X_test,
y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=0) columns
X_train.columnsos_data_X,os_data_y=os.fit_sample(X_train,                    =
y_train) os_data_X =
pd.DataFrame(data=os_data_X,columns=columns ) os_data_y=
pd.DataFrame(data=os_data_y,columns=['y'])
# we can Check the numbers of our data
print("length of oversampled data is ",len(os_data_X)) print("Number of no
subscription in oversampled data",len(os_data_y[os_data_y['y']==0]))
print("Number of subscription",len(os_data_y[os_data_y['y']==1]))
print("Proportion of no subscription data in oversampled data is
",len(os_data_y[os_data_y['y']==0])/len(os_data_X))
print("Proportion of subscription data in oversampled data is
```

```
",len(os_data_y[os_data_y['y']==1])/len(os_data_X))
length of oversampled data is  51134
Number of no subscription in oversampled data 25567
Number of subscription 25567
Proportion of no subscription data in oversampled data is  0.5
Proportion of subscription data in oversampled data is  0.5
```

Figure 16

Now we have a perfect balanced data! You may have noticed that I oversampled only on the training data, because by oversampling only on the training data, none of the information in the test data is being used to create synthetic observations, therefore, no information will bleed from test data into the model training.

Recursive Feature Elimination

Recursive Feature Elimination (RFE) is based on the idea to repeatedly construct a model and choose either the best or worst performing feature, setting the feature aside and then repeating the process with the rest of the features. This process is applied until all features in the dataset are exhausted. The goal of RFE is to select features by recursively considering smaller and smaller sets of features.

```
data_final_vars=data_final.columns.values.tolist()
y=['y']
X=[i for i in data_final_vars if i not in y]from sklearn.feature_selection import RFE
from sklearn.linear_model import LogisticRegressionlogreg =
LogisticRegression()rfe = RFE(logreg, 20)
rfe = rfe.fit(os_data_X, os_data_y.values.ravel())
print(rfe.support_) print(rfe.ranking_)
```

```
[False False False False False False False False  True False False  True
 False  True False False False False False False False False False False
 False  True False False False False  True False False False  True  True
 False False False False False False False  True  True  True  True  True
  True  True  True  True  True  True False False False False False False
  True False  True]
[39 38 26 42  9 12 24 36  1 35  8  1  7  1  5 32  2  4 31  3  6 10 23 21
 17  1 14 18 15 22  1 20 16 19  1  1 41 28 44 37 33 43 34  1  1  1  1  1
  1  1  1  1  1  1 29 30 11 27 40 25  1 13  1]
```

Figure 16

The RFE has helped us select the following features:

“euribor3m”, “job_bluecollar”, “job_housemaid”,

“marital_unknown”, “education_illiterate”, “default_no”,

“default_unknown”, “contact_cellular”, “contact_telephone”,

“month_apr”, “month_aug”, “month_dec”, “month_jul”, “month_jun”,

“month_mar”, “month_may”, “month_nov”, “month_oct”,

“poutcome_failure”, “poutcome_success”.

cols=['euribor3m', 'job_blue-collar', 'job_housemaid', 'marital_unknown',

'education_illiterate', 'default_no',
'default_unknown',

```
'month_apr',
'month_aug', 'month_dec', 'month_jul', 'month_jun',
'month_mar',
      'month_may', 'month_nov', 'month_oct',
"poutcome_failure",
"poutcome_success"]
X=os_data_X[cols]
y=os_data_y['y']
```

Implementing the model

```
import statsmodels.api as sm
logit_model=sm.Logit(y,X)
result=logit_model.fit()
print(result.summary2())
```

'contact_cellular', 'contact_telephone',

```
Warning: Maximum number of iterations has been exceeded.
         Current function value: 0.545891
         Iterations: 35
                                 Results: Logit
================================================================================
Model:                  Logit               No. Iterations:       35.0000
Dependent Variable:     y                   Pseudo R-squared:     0.212
Date:                   2018-09-10 12:16    AIC:                  55867.1778
No. Observations:       51134               BIC:                  56044.0219
Df Model:               19                  Log-Likelihood:       -27914.
Df Residuals:           51114               LL-Null:              -35443.
Converged:              0.0000              Scale:                1.0000
--------------------------------------------------------------------------------
                      Coef.    Std.Err.     z     P>|z|     [0.025     0.975]
--------------------------------------------------------------------------------
euribor3m            -0.4634    0.0091 -50.9471 0.0000     -0.4813    -0.4456
job_blue-collar      -0.1736    0.0283  -6.1230 0.0000     -0.2291    -0.1180
job_housemaid        -0.3260    0.0778  -4.1912 0.0000     -0.4784    -0.1735
marital_unknown       0.7454    0.2253   3.3082 0.0009      0.3038     1.1870
education_illiterate  1.3156    0.4373   3.0084 0.0026      0.4585     2.1727
default_no           16.1521 5414.0744   0.0030 0.9976 -10595.2387 10627.5429
default_unknown      15.8945 5414.0744   0.0029 0.9977 -10595.4963 10627.2853
contact_cellular    -13.9393 5414.0744  -0.0026 0.9979 -10625.3302 10597.4515
contact_telephone   -14.0065 5414.0744  -0.0026 0.9979 -10625.3973 10597.3843
month_apr            -0.8356    0.0913  -9.1490 0.0000     -1.0145    -0.6566
month_aug            -0.6882    0.0929  -7.4053 0.0000     -0.8703    -0.5061
month_dec            -0.4233    0.1655  -2.5579 0.0105     -0.7477    -0.0990
month_jul            -0.4056    0.0935  -4.3391 0.0000     -0.5889    -0.2224
month_jun            -0.4817    0.0917  -5.2550 0.0000     -0.6614    -0.3021
month_mar             0.6638    0.1229   5.3989 0.0000      0.4228     0.9047
month_may            -1.4752    0.0874 -16.8815 0.0000     -1.6465    -1.3039
month_nov            -0.8298    0.0942  -8.8085 0.0000     -1.0144    -0.6451
month_oct             0.5065    0.1175   4.3111 0.0000      0.2762     0.7367
poutcome_failure     -0.5000    0.0363 -13.7706 0.0000     -0.5711    -0.4288
poutcome_success      1.5788    0.0618  25.5313 0.0000      1.4576     1.7000
================================================================================
```

Figure 17

The p-values for most of the variables are smaller than 0.05, except four variables, therefore, we will remove them.

```
cols=['euribor3m', 'job_blue-collar', 'job_housemaid',
'marital_unknown', 'education_illiterate',
     'month_apr', 'month_aug', 'month_dec', 'month_jul', 'month_jun', 'month_mar',
     'month_may', 'month_nov', 'month_oct', "poutcome_failure", "poutcome_success"]
X=os_data_X[cols]
y=os_data_y['y']logit_model=sm.Logit(y,X) result=logit_model.fit()
print(result.summary2())
```

```
Optimization terminated successfully.
         Current function value: 0.555865
         Iterations 7
                          Results: Logit
=================================================================
Model:                Logit             No. Iterations:    7.0000
Dependent Variable:   y                 Pseudo R-squared:  0.198
Date:                 2018-09-10 12:38  AIC:               56879.2425
No. Observations:     51134             BIC:               57020.7178
Df Model:             15                Log-Likelihood:    -28424.
Df Residuals:         51118             LL-Null:           -35443.
Converged:            1.0000            Scale:             1.0000
-----------------------------------------------------------------
                      Coef.  Std.Err.    z     P>|z|   [0.025  0.975]
-----------------------------------------------------------------
euribor3m            -0.4488   0.0074 -60.6837 0.0000 -0.4633 -0.4343
job_blue-collar      -0.2060   0.0278  -7.4032 0.0000 -0.2605 -0.1515
job_housemaid        -0.2784   0.0762  -3.6519 0.0003 -0.4278 -0.1290
marital_unknown       0.7619   0.2244   3.3956 0.0007  0.3221  1.2017
education_illiterate  1.3080   0.4346   3.0096 0.0026  0.4562  2.1598
month_apr             1.2863   0.0380  33.8180 0.0000  1.2118  1.3609
month_aug             1.3959   0.0411  33.9688 0.0000  1.3153  1.4764
month_dec             1.8084   0.1441  12.5483 0.0000  1.5259  2.0908
month_jul             1.6747   0.0424  39.5076 0.0000  1.5916  1.7578
month_jun             1.5574   0.0408  38.1351 0.0000  1.4773  1.6374
month_mar             2.8215   0.0908  31.0891 0.0000  2.6437  2.9994
month_may             0.5848   0.0304  19.2166 0.0000  0.5251  0.6444
month_nov             1.2725   0.0445  28.5720 0.0000  1.1852  1.3598
month_oct             2.7279   0.0816  33.4350 0.0000  2.5680  2.8878
poutcome_failure     -0.2797   0.0351  -7.9753 0.0000 -0.3485 -0.2110
poutcome_success      1.9617   0.0602  32.5939 0.0000  1.8438  2.0797
=================================================================
```

Figure 18

Logistic Regression Model Fitting

from sklearn.linear_model import LogisticRegression

Figure 19

Predicting the test set results and calculating the accuracy

```
y_pred =      logreg.predict(X_test)
print('Accuracy of logistic regression
```

classifier on test set: {:.2f}'.format(logreg.score(X_test, y_test)))

Accuracy of logistic regression classifier on test set: 0.74

Confusion Matrix

```
from sklearn.metrics import confusion_matrix
confusion_matrix = confusion_matrix(y_test, y_pred)
print(confusion_matrix)
```

[[6124 1542]

[2505 5170]]

The result is telling us that we have ***6124+5170*** correct predictions and ***2505+1542*** incorrect predictions.

```
from sklearn import metricsX_train, X_test, y_train, y_test =
train_test_split(X, y, test_size=0.3, random_state=0)
logreg = LogisticRegression()
logreg.fit(X_train, y_train)
```

```
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
          intercept_scaling=1, max_iter=100, multi_class='ovr', n_jobs=1,
          penalty='l2', random_state=None, solver='liblinear', tol=0.0001,
          verbose=0, warm_start=False)
```

Compute precision, recall, F-measure and support

To quote from Scikit Learn:

The precision is the ratio tp / (tp + fp) where tp is the number of true positives and fp the number of false positives. The precision is intuitively the ability of the classifier to not label a sample as positive if it is negative.

The recall is the ratio tp / (tp + fn) where tp is the number of true positives and fn the number of false negatives. The recall is intuitively the ability of the classifier to find all the positive samples.

The F-beta score can be interpreted as a weighted harmonic mean of the precision and recall, where an F-beta score reaches its best value at 1 and

The support is the number of occurrences of each class in y_test.

	precision	recall	f1-score	support
0	0.71	0.80	0.75	7666
1	0.77	0.67	0.72	7675
avg / total	0.74	0.74	0.74	15341

worst score at 0.

The F-beta score weights the recall more than the precision by a factor of beta. beta = 1.0 means recall and precision are equally important.

Figure 20

Interpretation: Of the entire test set, 74% of the promoted term deposit were the term deposit that the customers liked. Of the entire test set, 74% of the customer's preferred term deposits that were promoted.

ROC Curve

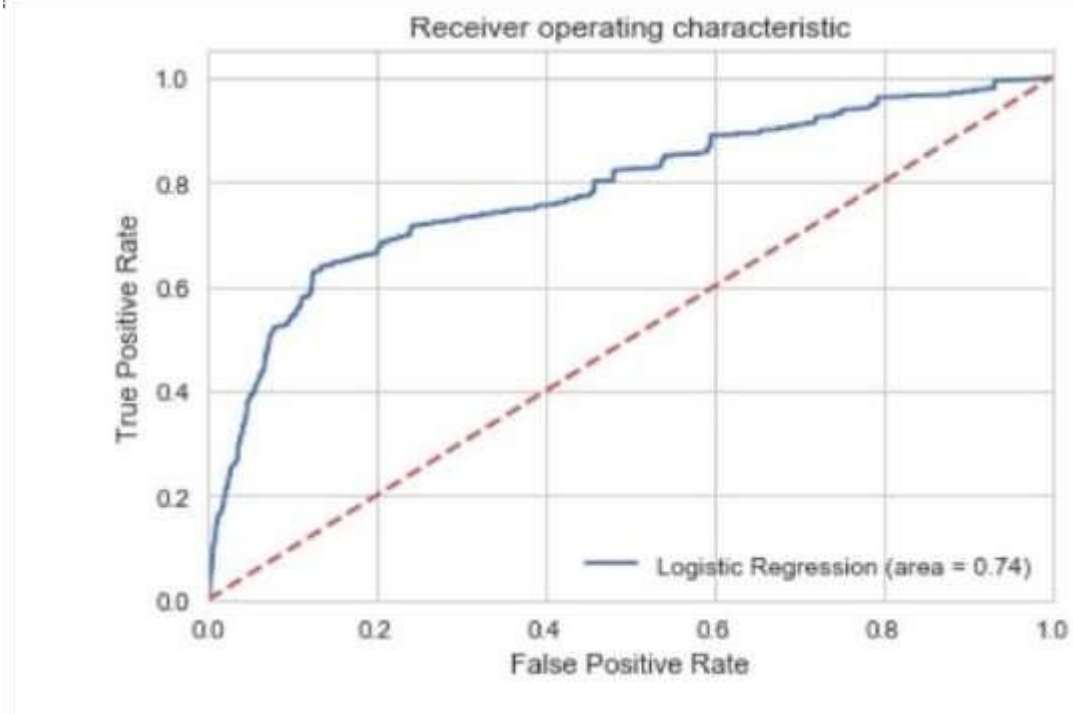
```
from sklearn.metrics import roc_auc_score from sklearn.metrics import
roc_curve  logit_roc_auc = roc_auc_score(y_test, logreg.predict(X_test)) fpr,
tpr, thresholds = roc_curve(y_test, logreg.predict_proba(X_test)[:,1])
plt.figure()
plt.plot(fpr, tpr, label='Logistic Regression (area = %0.2f)' % logit_roc_auc)
plt.plot([0, 1], [0, 1],'r--') plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
```

Figure 21

The receiver operating characteristic (ROC) curve is another common tool used with binary classifiers. The dotted line represents the ROC curve of a purely random classifier; a good classifier stays as far away from that line as possible (toward the top-left corner).

The Jupyter notebook used to make this post is available here. I would be

```
plt.ylabel('True Positive Rate')
plt.title('Receiver operating characteristic')
plt.legend(loc="lower right")
plt.savefig('Log_ROC')
plt.show()
```

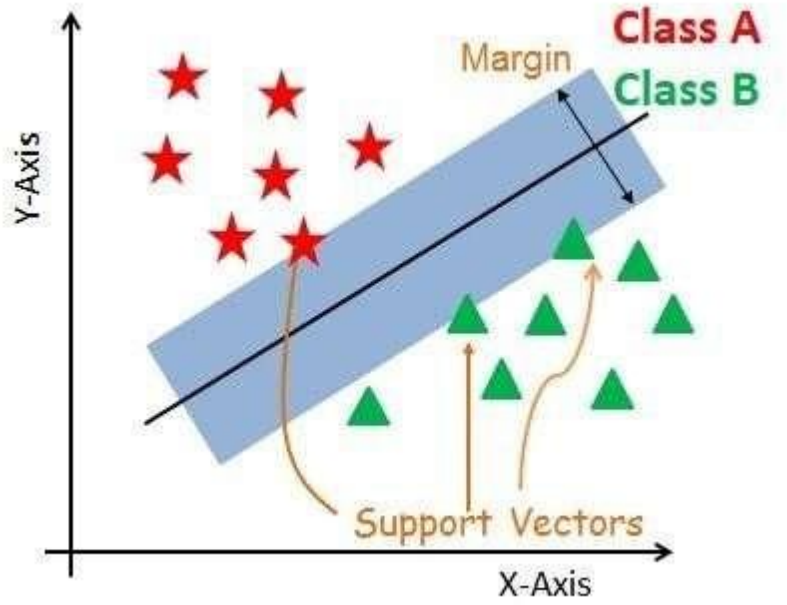


pleased to receive feedback or questions on any of the above.

Reference: Learning Predictive Analytics with Python book

Support Vector Machines

Generally, Support Vector Machines is considered to be a classification approach, it but can be employed in both types of classification and regression problems. It can easily handle multiple continuous and categorical variables. SVM constructs a hyperplane in multidimensional space to separate different classes. SVM generates optimal hyperplane in an iterative manner, which is used to minimize an error. The core idea of SVM is to find a maximum marginal hyperplane(MMH) that best divides the dataset into classes.



Support Vectors

Support vectors are the data points, which are closest to the hyperplane. These points will define the separating line better by calculating margins. These points are more relevant to the construction of the classifier.

Hyperplane

A hyperplane is a decision plane which separates between a set of objects having different class memberships.

Margin

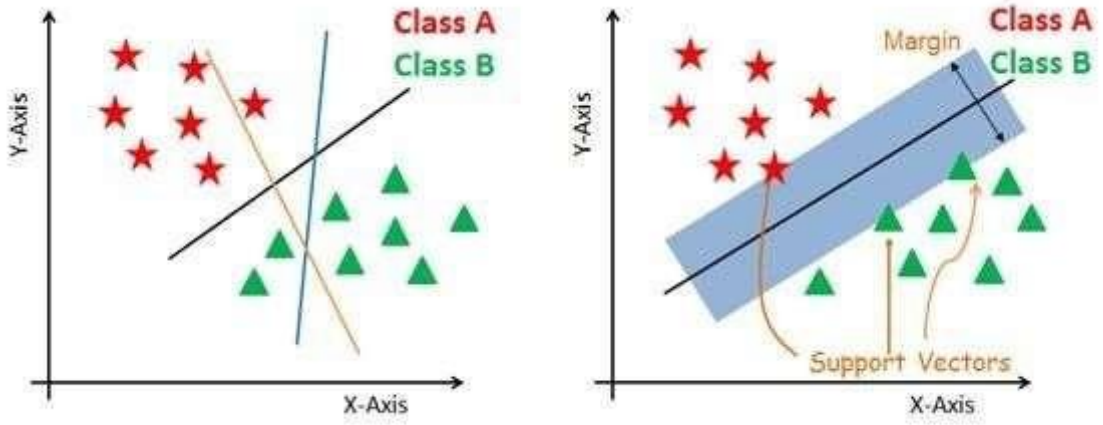
A margin is a gap between the two lines on the closest class points. This is calculated as the perpendicular distance from the line to support vectors or closest points. If the margin is larger in between the classes, then it is considered a good margin, a smaller margin is a bad margin.

How does SVM work?

The main objective is to segregate the given dataset in the best possible way. The distance between the either nearest points is known as the margin. The objective is to select a hyperplane with the maximum possible margin between support vectors in the given dataset. SVM searches for the maximum marginal hyperplane in the following steps:

1. Generate hyperplanes which segregates the classes in the best way. Left-hand side figure showing three hyperplanes black, blue and orange. Here, the blue and orange have higher classification error, but the black is separating the two classes correctly.

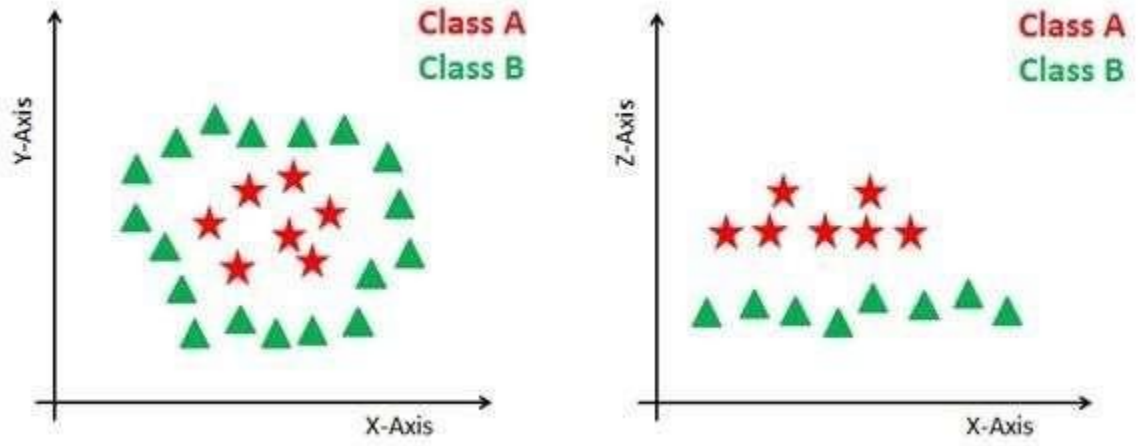
2. Select the right hyperplane with the maximum segregation from the either nearest data points as shown in the right-hand side figure.



Dealing with non-linear and inseparable planes

Some problems can't be solved using linear hyperplane, as shown in the figure below (left-hand side).

In such situation, SVM uses a kernel trick to transform the input space to a higher dimensional space as shown on the right. The data points are plotted on the x-axis and z-axis (Z is the squared sum of both x and y: z=x^2=y^2). Now you can easily segregate these points using linear separation.



SVM Kernels

The SVM algorithm is implemented in practice using a kernel. A kernel transforms an input data space into the required form. SVM uses a technique called the kernel trick. Here, the kernel takes a low-dimensional input space and transforms it into a higher dimensional space. In other words, you can say that it converts nonseparable problem to separable problems by adding more dimension to it. It is most useful in non-linear separation problem. Kernel trick helps you to build a more accurate classifier.

- **Linear Kernel** A linear kernel can be used as normal dot product any two given observations.
  The product between two vectors is the sum of the multiplication of each pair of input values.

```
K x  xi     sum x    xi
```

- **Polynomial Kernel** A polynomial kernel is a more generalized form of the linear kernel. The polynomial kernel can distinguish curved or nonlinear input space.

```
K x xi     1    sum x
xi   d
```


Where d is the degree of the polynomial. d=1 is similar to the linear transformation. The degree needs to be manually specified in the learning algorithm.

- **Radial Basis Function Kernel** The Radial basis function kernel is a popular kernel function commonly used in support vector machine classification. RBF can map an input space in infinite dimensional space.

```
K x xi     exp   gamma    sum   x – xi  2
```


Here gamma is a parameter, which ranges from 0 to 1. A higher value of gamma will perfectly fit the training dataset, which causes over-fitting. Gamma=0.1 is considered to be a good default value. The value of gamma needs to be manually specified in the learning algorithm.

Classifier Building in Scikit-learn

Until now, you have learned about the theoretical background of SVM. Now you will learn about its implementation in Python using scikit-learn.

In the model the building part, you can use the cancer dataset, which is a very famous multi-class classification problem. This dataset is computed from a digitized image of a fine needle aspirate (FNA) of a breast mass. They describe characteristics of the cell nuclei present in the image.

The dataset comprises 30 features (mean radius, mean texture, mean perimeter, mean area, mean smoothness, mean compactness, mean concavity, mean concave points, mean symmetry, mean fractal dimension, radius error, texture error, perimeter error, area error, smoothness error,

compactness error, concavity error, concave points error, symmetry error, fractal dimension error, worst radius, worst texture, worst perimeter, worst area, worst smoothness, worst compactness, worst concavity, worst concave points, worst symmetry, and worst fractal dimension) and a target (type of cancer).

This data has two types of cancer classes: malignant (harmful) and benign (not harmful). Here, you can build a model to classify the type of cancer. The dataset is available in the scikit-learn library or you can also download it from the UCI Machine Learning Library.

Loading Data

Let's first load the required dataset you will use.

```
#Import scikit-learn

dataset library from

sklearn        import

datasets

#Load dataset

cancer =

datasets.load_brea

st_cancer()
```

Exploring Data

After you have loaded the dataset, you might want to know a little bit more about it. You can check feature and target names.

```
# print the names of the 13

features print("Features: ",

cancer.feature_names)

# print the label type of cancer('malignant'

'benign') print("Labels: ", cancer.target_names)
```

```
Features: ['mean radius' 'mean texture' 'mean perimeter' 'mean area'

 'mean smoothness' 'mean compactness' 'mean concavity'
```

'mean concave points' 'mean symmetry' 'mean fractal dimension'

'radius error' 'texture error' 'perimeter error' 'area error'

'smoothness error' 'compactness error' 'concavity error'

'concave points error' 'symmetry error' 'fractal dimension error'

'worst radius' 'worst texture' 'worst perimeter' 'worst area'

'worst smoothness' 'worst compactness' 'worst concavity'

'worst concave points' 'worst symmetry' 'worst fractal dimension']

Labels: ['malignant' 'benign']

Let's explore it for a bit more. You can also check the shape of the dataset using shape.

```
# print data(feature)shape



cancer.data.shape
```

POWER ED B Y DAT AC AMP WORK SPA CE COPY CODE

```
569
30
```

POWER ED B Y DAT AC AMP WORK SPA CE COPY CODE

Let's check top 5 records of the feature set.

```
# print the cancer data features (top 5 records)



print(cancer.data[0:5])
```

POWER ED B Y DAT AC AMP WORK SPA CE COPY CODE

```
 1.799e+01  1.038e+01  1.228e+02  1.001
3.001e-01

 1.471e-01  2.419e-01  7.871e-02  1.095e+
1.534e+02                                  6.399e-

03 4.904e-02  5.373e-02  1.587e-02
2.538e+01

 1.733e+01  1.846e+02  2.019e+03  1.622e
2.654e-01

 4.601e-01  1.189e
01

[2.057e+01       1.777e+01       1.329e+02
1.326e+03 8.474e-02 7.864e-02 8.690e-02

 7.017e-02  1.812e-01  5.667e-02  5.435e-
01 7.339e-01 3.398e+00 7.408e+01

 5.225e-03  1.308e-02  1.860e-02  1.340e-
2.499e+01

 2.341e+01  1.588e+02  1.956e+03  1.238e
1.860e-01

 2.750e-01  8.902e
02

[1.969e+01       2.125e+01       1.300e+02
1.203e+03 1.096e-01 1.599e-01 1.974e-01
```

1.279e-01 2.069e-01 5.999e-02 7.456e-
01 7.869e-01 4.585e+00 9.403e+01

6.150e-03 4.006e-02 3.832e-02 2.058e-
2.357e+01

2.553e+01 1.525e+02 1.709e+03 1.444e
2.430e-01

3.613e-01 8.758e
02

[1.142e+01 2.038e+01 7.758e+01
3.861e+02 1.425e-01 2.839e-01 2.414e-01

1.052e-01 2.597e-01 9.744e-02 4.956e- 9.110e-03 7.458e-02
2.723e+01 5.661e-

02 1.867e-02 5.963e-02
1.491e+01

2.650e+01 9.887e+01 5.677e+02 2.098e
2.575e-01

6.638e-01 1.730e
01

[2.029e+01 1.434e+01 1.351e+02
1.297e+03 1.003e-01 1.328e-01 1.980e-01

1.043e-01 1.809e-01 5.883e-02 7.572e-
01 7.813e-01 5.438e+00 9.444e+01

```
 1.149e-02  2.461e-02  5.688e-02  1.885e-0
2.254e+01


 1.667e+01  1.522e+02  1.575e+03  1.374e  2.364e-01   7.678e-
1.625e-01                                  02
```

POWER ED B Y DAT AC AMP WORK SPA CE COPY CODE

Let's take a look at the target set.

```
# print the cancer labels (0:m
1:benign)


print(cancer.target)
```

POWER ED B Y DAT AC AMP WORK SPA CE COPY CODE

```
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 0 0
0 0 0 0 0


1 0 0 0 0 0 0 0 0 1 0 1 1 1 1 1 0 0 1 0 0 1 1 1
1 0 1 0 0


1 0 1 0 0 1 1 1 0 0 1 0 0 0 1 1 1 0 1 1 0 0 1 1
1 1 0 1 1


1 1 1 1 1 1 0 0 0 1 0 0 1 1 1 0 0 1 0 1 0 0 1
1 1 1 0 1


1 1 1 1 1 1 1 1 0 1 1 1 1 0 0 1 0 1 1 0 0 1 1
0 0 1 1 1 1 0 1 1 0 0 0 1 0


1 0 1 1 1 0 1 1 0 0 1 0 0 0 0 1 0 0 0 1 0 1 0
1 1 0 1 0 0 0 0 1 1 0 0 1 1
```

```
1 0 1 1 1 1 1 0 0 1 1 0 1 1 0 0 1 0 1 1 1 1 0
1 1 1 1 1 0 1 0 0 0 0 0 0 0

0 0 0 0 0 0 0 1 1 1 1 1 1 0 1 0 1 1 0 1 1 0 1
1 1 1 1 1

1 0 1 1 0 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1
0 0 0 1 1

1 1 0 1 0 1 0 1 1 1 0 1 1 1 1 1 1 1 0 0 0 1 1
1 1 1 1 1 1 1 1 1 1 0 0 1 0 0

0 1 0 0 1 1 1 1 1 0 1 1 1 1 1 0 1 1 1 0 1 1 0
1 1 1 1 1

1 0 1 1 1 1 1 0 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1
1 1 0 1 1

0       1 0 1 1 0 1 0 1 1 1 1 1 1 1 1 0 0 1 1 1
        1 1 1 0 1

1       1 1 1 1 1 1 0 1 0 1 1 0 1 1 1 1 1 1 0 0 1 0
        1 0 1 0 0                                     1 1 1 0 1 1 1
                                                      1

1 1 1 1 1 1 1 0 1 0 0 1 1 1 1 1 1 1 1
1 1 1 1

1 1 1 1 1 1 1 1 0 0 0 0 0
0 1
```

POWER ED B Y DAT AC AMP WORK SPA CE COPY CODE

Splitting Data

```
from sklearn.model_selection import train_test_split

# Split dataset into training set and test
set

X_train, X_test, y_train, y_test = train_test_split(cancer.data, cancer.target,
test_size=0.3,random_state=109) #
70% training and 30% test
```


To understand model performance, dividing the dataset into a training set and a test set is a good strategy.

Split the dataset by using the function `train_test_split()`. you need to pass 3 parameters features, target, and test_set size. Additionally, you can use random_state to select records randomly.

```
# Import train_test_split function
```

Generating Model

Let's build support vector machine model. First, import the SVM module and create support vector classifier object by passing argument kernel as the linear kernel in `SVC()` function.

Then, fit your model on train set using `fit()` and perform prediction on the test set using `predict()`.

```
#Import svm model

from sklearn import svm

#Create a svm Classifier

clf = svm.SVC(kernel='linear') # Linear Kernel #Train

the model using the training sets

clf.fit(X_train, y_train)

#Predict the response for test dataset

y_pred = clf.predict(X_test)
```

POWER ED B Y DAT AC AMP WORK SPA CE COPY CODE

Evaluating the Model

Let's estimate how accurately the classifier or model can predict the breast cancer of patients.

Accuracy can be computed by comparing actual test set values and predicted values.

```
#Import scikit-learn metrics module for accuracy calculation

from sklearn import
metrics

# Model Accuracy: how often is the classifier
correct?

print("Accuracy:",metrics.accuracy_score(y_test,
y_pred))
```

```
Accuracy 0.9649122807017544
```

Well, you got a classification rate of 96.49%, considered as very good accuracy.

For further evaluation, you can also check precision and recall of model.

```
# Model Precision: what percentage of positive tuples are labeled as such?
print("Precision:",metrics.precision_score(y_test, y_pred))
# Model Recall: what percentage of positive tuples are labe lled as such?
print("Recall:",metrics.recall_score(y_test,
y_pred))
```

```
Precision :  0.9811320754716981
Recall 0.9629629629629629
```

SVM Applications in Real World

As we have seen in the previous article, Support Vector Machine is a really powerful Supervised Machine Learning Algorithm. The algorithm is quite flexible and provides us with effective results. Though SVMs are capable of handling both regression and classification problems, we mainly use SVM for classification problems. In this article, we are going to have a brief introduction to some of the most widely spread applications of SVM. Let's start!!!

SVM Applications

1. Face Detection

As the name suggests, the problem deals with identifying facial features of the image that we give in as the input. In this type of problem, the algorithm tends to classify the objects in the image as facial and nonfacial. For the training of SVM to deal with this type of problem, we train the model with n x n pixels. After processing, we classify this data as facial (storing +1) and non-facial (storing -1).

The algorithm processes each pixel of the input image and classifies it to be facial and nonfacial. We can even highlight the faces seen in the images by fabricating a box-like boundary around the face. With this, users can segregate the face from the rest of the image.

2. Text and Hypertext Classification

We can classify both types of data, namely, inductive and transductive using Support Vector Machine classification for text and hypertext. We train the model using labeled data that we already segregate as new articles, e-mails, blogs, web pages, and so on. As soon as we pass our input text to the model, the trained model tends to classify the given text into one of the predefined classes. Some of the examples of text classification are:

Classifying news into various categories like “Sports”, “Business”, and “Entertainment”.

Classifying web pages into categories like personal pages, blogs, business sites, and so on.

3. Classification of Images

When we use image classifying algorithms along with SVM, we can get accurate results for the classification of the input images. We train the model in such a way that it extracts useful features from the image and stores it for classification. The distinct features help the model in providing many accurate classifying results. SVM algorithms along with image processing algorithms provide highly accurate results as compared to conventional querybased reinforcement schemes.

4. Stenographic Detection in Digital Images

SVM models are capable of judging the input images to verify if the images are pure or if they are adulterated with filters. This helps the security-based organizations to look up any hidden messages and other information. We can easily reveal the encrypted information in the image with the help of SVM.

When we deal with high-resolution images, the images contain a higher level of pixelation that provides a convenient environment for image encryption. However, with Support Vector Models, we can easily decipher these encrypted messages. SVM provides reliable and accurate results even with varying datasets and is able to analyze this encrypted information accurately.

5. Protein Fold and Remote Homology Detection

For developers in today’s world, Protein remote homology proves to be a primary problem in the field of computational biology. The most reliable and widely used solution to cater to this problem is the Support Vector

Machine. In recent years, we have seen a massive surge in the usage of SVM for the detection of protein remote homology.

SVM is capable of identifying a wide range of biological sequences. The results of these algorithms depend mostly on the architecture of these protein sequences. This further helps us to classify genes and thus diagnose ailments with more precision.

6. **Handwriting Recognition** :SVM proves to be a really good tool for recognizing handwritings from the given input dataset. This helps security agencies in authorizing users and providing better security to the users. It is also helpful in data entry and signature validation for accounting issues.

ENSEMBLE LEARNING

Ensemble means a group of elements viewed as a whole rather than individually. An Ensemble method creates multiple models and combines them to solve it. Ensemble methods help to improve the robustness/generalizability of the model. In this article, we will discuss some methods with their implementation in Python. For this, we choose a **dataset** from the UCI repository.

Introduction to Ensemble Learning

Let's start with an example, you have a question, and you ask it around, then aggregate their answers. In most cases, you would find that this answer is way better than one expert's answer. So this is called the **wisdom of the crowd**. Similarly, if you aggregate the predictions of models such as classifiers or regressors, you would notice that the group has better performance than the best individual model. So this group of predictors or models are called **ensemble** and hence this technique is knowns as **Ensemble learning**.For example, you can train a group of decision trees classifiers, on different random subsets of the training data. After that to make predictions, obtain predictions from all individual trees and predict the most frequent class (i.e predicted the most). This ensemble of decision trees is called **Random Forest** and is one of the most powerful algorithms in the machine learning world.You will often use this technique, at the end of a project. Once you have a couple of good predictors in your hand and combine them even o a better one. Most of the winning solutions in machine

learning competitions involve ensemble methods. (For example- Netflix's prize competition).So, go grab a coffee, make yourself comfortable.

And let's begin our journey of learning about Ensemble Learning

Basic ensemble methods

1. Averaging method: It is mainly used for regression problems. The method consists of building multiple models independently and returning the average of the prediction of all the models. In general, the combined output is better than an individual output because variance is reduced.

In the below example, three regression models (linear regression, xgboost, and random forest) are trained and their predictions are averaged. The final prediction output is pred_final.

Python3

```
# importing utility modules import pandas as pd from
sklearn.model_selection import train_test_split from
sklearn.metrics import mean_squared_error
# importing machine learning models for prediction from
sklearn.ensemble import RandomForestRegressor
import xgboost as xgb from sklearn.linear_model import
LinearRegression
# loading train data set in dataframe from train_data.csv file df =
pd.read_csv("train_data.csv")
# getting target data from the dataframe target =
df["target"]
# getting train data from the dataframe train =
df.drop("target")
```

```
# Splitting between train data into training and validation dataset
X_train, X_test, y_train, y_test = train_test_split( train, target,
test_size=0.20)
# initializing all the model objects with default parameters model_1 =
LinearRegression()  model_2  =  xgb.XGBRegressor()  model_3  =
RandomForestRegressor()
# training all the model on the training dataset
model_1.fit(X_train,                    y_target)
model_2.fit(X_train,                    y_target)
model_3.fit(X_train, y_target)
# predicting the output on the validation dataset pred_1 =
model_1.predict(X_test) pred_2 =
model_2.predict(X_test)                 pred_3          =
model_3.predict(X_test)
# final prediction after averaging on the prediction of all 3 models
pred_final = (pred_1+pred_2+pred_3)/3.0
# printing the root mean squared error between real value and predicted
value print(mean_squared_error(y_test, pred_final))
```

Output:
4560

2. Max voting: It is mainly used for classification problems. The method consists of building multiple models independently and getting their individual output called 'vote'.

The class with maximum votes is returned as output. In the below example, three classification models (logistic regression, xgboost, and random forest) are combined using sklearn VotingClassifier, that model is trained and the class with maximum votes is returned as output. The final prediction output is pred_final. Please note it's a classification, not regression, so the loss may be different from other types of ensemble methods.

```
# importing utility modules import pandas as pd from
sklearn.model_selection import train_test_split from
sklearn.metrics import log_loss
# importing machine learning models for prediction from
sklearn.ensemble import RandomForestClassifier
from xgboost import XGBClassifier from
sklearn.linear_model import LogisticRegression
# importing voting classifier from
sklearn.ensemble import VotingClassifier
# loading train data set in dataframe from train_data.csv file
df = pd.read_csv("train_data.csv")
# getting target data from the dataframe target
= df["Weekday"]
```

```
# getting train data from the dataframe train =
df.drop("Weekday")
# Splitting between train data into training and validation dataset
X_train, X_test, y_train, y_test = train_test_split( train, target,
test_size=0.20)
# initializing all the model objects with default parameters model_1 =
LogisticRegression() model_2 = XGBClassifier() model_3 =
RandomForestClassifier()
# Making the final model using voting classifier final_model
= VotingClassifier( estimators=[('lr', model_1), ('xgb',
model_2), ('rf', model_3)], voting='hard')
# training all the model on the train dataset final_model.fit(X_train,
y_train)
# predicting the output on the test dataset pred_final =
final_model.predict(X_test)
# printing log loss between actual and predicted value
print(log_loss(y_test, pred_final))
```

Output:
231

Let's have a look at a bit more advanced ensemble methods **Advanced ensemble methods**

Ensemble methods are extensively used in classical machine learning. Examples of algorithms using bagging are random forest and bagging metaestimator and examples of algorithms using boosting are GBM, XGBM, Adaboost, etc.

As a developer of a machine learning model, it is highly recommended to use ensemble methods. The ensemble methods are used extensively in almost all competitions and research papers.

1. Stacking: It is an ensemble method that combines multiple models (classification or regression) via meta-model (metaclassifier or metaregression). The base models are trained on the complete dataset, then the meta-model is trained on features returned (as output) from base models. The base models in stacking are typically different. The meta-model helps to find the features from base models to achieve the best accuracy.

Algorithm:

1. *Split the train dataset into n parts*
2. *A base model (say linear regression) is fitted on n-1 parts and predictions are made for the nth part. This is done for each one of the n part of the train set.*
3. *The base model is then fitted on the whole train dataset.*
4. *This model is used to predict the test dataset.*
5. *The Steps 2 to 4 are repeated for another base model which results in another set of predictions for the train and test dataset.*
6. *The predictions on train data set are used as a feature to build the new model.*
7. *This final model is used to make the predictions on test dataset*
   Stacking is a bit different from the basic ensembling methods because it has first-level and second-level models. Stacking features are first extracted by training the dataset with all the first-level models. A first-level model is then using the train stacking features to train the model than this model predicts the final output with test stacking features.

Python3

```
#     importing
utility modules
import  pandas
as pd
```

```
from sklearn.model_selection import train_test_split from
sklearn.metrics import mean_squared_error
# importing machine learning models for prediction from
sklearn.ensemble import RandomForestRegressor
import xgboost as xgb from sklearn.linear_model import
LinearRegression
# importing stacking lib from vecstack
import stacking
# loading train data set in dataframe from train_data.csv file df =
pd.read_csv("train_data.csv")
# getting target data from the dataframe target =
df["target"]
# getting train data from the dataframe train =
df.drop("target")
# Splitting between train data into training and validation dataset
X_train, X_test, y_train, y_test = train_test_split( train, target,
test_size=0.20)


# initializing all the base model objects with default parameters
```

```
model_1 = LinearRegression() model_2 =
xgb.XGBRegressor() model_3 =
RandomForestRegressor()
# putting all base model objects in one list all_models =
[model_1, model_2, model_3]   # computing the stack
features s_train, s_test = stacking(all_models, X_train,
X_test,                    y_train, regression=True,
n_folds=4)
# initializing the  second-level  model  final_model  =
model_1
# fitting the second level model with stack features final_model =
final_model.fit(s_train, y_train)
# predicting the  final  output  using  stacking  pred_final  =
final_model.predict(X_test)
# printing the root mean squared error between real value and predicted
value print(mean_squared_error(y_test, pred_final))
```

Output:
4510

2. Blending: It is similar to the stacking method explained above, but rather than using the whole dataset for training the base-models, a validation dataset is kept separate to make predictions.

Algorithm:

1. *Split the training dataset into train, test and validation dataset.*
2. *Fit all the base models using train dataset.*
3. *Make predictions on validation and test dataset.*

4. *These predictions are used as features to build a second level model 5. This model is used to make predictions on test and meta-features*

Python3

```
# importing utility modules import pandas as pd

from sklearn.metrics import mean_squared_error

# importing machine learning models for prediction

from sklearn.ensemble import RandomForestRegressor

import xgboost as xgb from sklearn.linear_model

import LinearRegression

# importing train test split from

sklearn.model_selection import train_test_split

# loading train data set in dataframe from train_data.csv file

df = pd.read_csv("train_data.csv")

# getting target data from the dataframe

target = df["target"]

# getting train data from the dataframe

train = df.drop("target")

#Splitting between train data into training and validation
dataset
```

```
X_train, X_test, y_train, y_test = train_test_split(train, target,
test_size=0.20)
# performing the train test and validation split
train_ratio = 0.70 validation_ratio = 0.20 test_ratio =
0.10
# performing train test split x_train, x_test, y_train, y_test =
train_test_split(     train, target, test_size=1 - train_ratio)
# performing test validation split x_val, x_test, y_val, y_test
= train_test_split(      x_test, y_test,
test_size=test_ratio/(test_ratio + validation_ratio))
# initializing all the base model objects with default
parameters model_1 = LinearRegression() model_2 =
xgb.XGBRegressor() model_3 = RandomForestRegressor()
# training all the model on the train dataset
# training first model model_1.fit(x_train, y_train)
val_pred_1 = model_1.predict(x_val)
```

```
test_pred_1 = model_1.predict(x_test)
# converting to dataframe val_pred_1 = pd.DataFrame(val_pred_1)
        test_pred_1    =
pd.DataFrame(test_pred_1)
#  training  second  model  model_2.fit(x_train,
y_train)  val_pred_2  =  model_2.predict(x_val)
test_pred_2 = model_2.predict(x_test)
# converting to dataframe val_pred_2 = pd.DataFrame(val_pred_2)
        test_pred_2    =
pd.DataFrame(test_pred_2)
# training third model model_3.fit(x_train, y_train)
val_pred_3 = model_1.predict(x_val) test_pred_3
= model_1.predict(x_test)
# converting to dataframe val_pred_3 = pd.DataFrame(val_pred_3)
        test_pred_3    =
pd.DataFrame(test_pred_3)
# concatenating validation dataset along with all the predicted
validation data (meta features)
df_val = pd.concat([x_val, val_pred_1, val_pred_2, val_pred_3],
axis=1)
```

```
df_test = pd.concat([x_test, test_pred_1, test_pred_2, test_pred_3],
axis=1)
# making the final model using the meta features final_model      =
      LinearRegression() final_model.fit(df_val, y_val)
# getting the final output final_pred = final_model.predict(df_test)
```

```
#printing the root mean squared error between real value and
predicted      value   print(mean_squared_error(y_test,
pred_final))
```

Output:
4790

3. Bagging: It is also known as a bootstrapping method. Base models are run on bags to get a fair distribution of the whole dataset. A bag is a subset of the dataset along with a replacement to make the size of the bag the same as the whole dataset. The final output is formed after combining the output of all base models.

Algorithm:

1. *Create multiple datasets from the train dataset by selecting observations with replacements*
2. *Run a base model on each of the created datasets independently*
3. *Combine the predictions of all the base models to each the final output* Bagging normally uses only one base model (XGBoost Regressor used in the code below).

Python

```
#     importing
utility modules
import  pandas
as pd
```

```
from sklearn.model_selection import train_test_split from
sklearn.metrics import mean_squared_error
# importing machine learning models for prediction import
xgboost as xgb
# importing bagging module from
sklearn.ensemble import BaggingRegressor
# loading train data set in dataframe from train_data.csv file df =
pd.read_csv("train_data.csv")
# getting target data from the dataframe target =
df["target"]
# getting train data from the dataframe train =
df.drop("target")
# Splitting between train data into training and validation dataset
X_train, X_test, y_train, y_test = train_test_split( train, target,
test_size=0.20)
# initializing the bagging model using XGboost as base model with
default parameters model =
BaggingRegressor(base_estimator=xgb.XGBRegressor())
# training model
```

```
model.fit(X_train, y_train)
# predicting the output on the test dataset pred =
model.predict(X_test)
# printing the root mean squared error between real value and predicted
value print(mean_squared_error(y_test, pred_final))
```

Output:
4666

4. **Boosting:** Boosting is a sequential method–it aims to prevent a wrong base model from affecting the final output. Instead of combing the base models, the method focuses on building a new model that is dependent on the previous one. A new model tries to remove the errors made by its previous one. Each of these models is called weak learners. The final model (aka strong learner) is formed by getting the weighted mean of all the weak learners.
Algorithm:

1. *Take a subset of the train dataset.*
2. *Train a base model on that dataset.*
3. *Use third model to make predictions on the whole dataset.*
4. *Calculate errors using the predicted values and actual values.*
5. *Initialize all data points with same weight.*
6. *Assign higher weight to incorrectly predicted data points.*
7. *Make another model, make predictions using the new model in such a way that errors made by the previous model are mitigated/corrected.*
8. *Similarly, create multiple models–each successive model correcting the errors of the previous model.*
9. *The final model (strong learner) is the weighted mean of all the previous models (weak learners).*

Python3

```
# importing utilitymodules
import pandas as pd
```

```
from sklearn.model_selection import train_test_split from
sklearn.metrics import mean_squared_error
# importing machine learning models for prediction from
sklearn.ensemble import GradientBoostingRegressor
# loading train data set in dataframe from train_data.csv file df =
pd.read_csv("train_data.csv")
# getting target data from the dataframe target =
df["target"]
# getting train data from the dataframe train =
df.drop("target")
# Splitting between train data into training and validation dataset
X_train, X_test, y_train, y_test = train_test_split(  train, target,
test_size=0.20)
# initializing the boosting module with default parameters model =
GradientBoostingRegressor()
# training the model on the train dataset model.fit(X_train, y_train)
# predicting the output  on the test dataset pred_final =
model.predict(X_test)
# printing the root mean squared error between real value and
predicted value print(mean_squared_error(y_test, pred_final))
```

Output:

4789

Note: The scikit-learn provides several modules/methods for ensemble methods. Please note the accuracy of a method does not suggest one method is superior to another. The article aims to give a brief introduction to ensemble methods–not to compare between them. The programmer must use a method that suits the data.

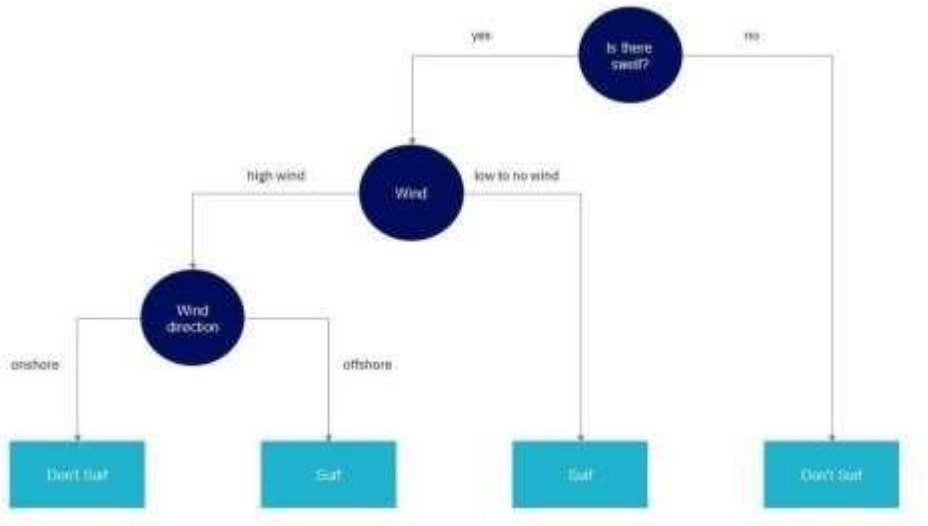
RANDO

M FOREST What is random forest?

Random forest is a commonly-used machine learning algorithm trademarked by Leo Breiman and Adele Cutler, which combines the output of multiple decision trees to reach a single result. Its ease of use and flexibility have fueled its adoption, as it handles both classification and regression problems.

Decision trees

Since the random forest model is made up of multiple decision trees, it would be helpful to start by describing the decision tree algorithm briefly. Decision trees start with a basic question, such as, “Should I surf?” From there, you can ask a series of questions to determine an answer, such as, “Is it a long period swell?” or “Is the wind blowing offshore?”. These questions make up the decision nodes in the tree, acting as a means to split the data. Each question helps an individual to arrive at a final decision, which would be denoted by the leaf node. Observations that fit the criteria will follow the “Yes” branch and those that don’t will follow the alternate path. Decision trees seek to find the best split to subset the data, and they are typically trained through the Classification and Regression Tree (CART) algorithm. Metrics, such as Gini impurity, information gain, or mean square error (MSE), can be used to evaluate the quality of the split.



This decision tree is an example of a classification problem, where the class labels are "surf" and "don't surf."

While decision trees are common supervised learning algorithms, they can be prone to problems, such as bias and overfitting. However, when multiple decision trees form an ensemble in the random forest algorithm, they predict more accurate results, particularly when the individual trees are uncorrelated with each other.

Ensemble methods

Ensemble learning methods are made up of a set of classifiers—e.g. decision trees—and their predictions are aggregated to identify the most popular result. The most well-known ensemble methods are bagging, also known as bootstrap aggregation, and boosting. In 1996, Leo Breiman (link resides outside IBM) (PDF, 810 KB) introduced the bagging method; in this method, a random sample of data in a training set is selected with replacement—meaning that the individual data points can be chosen more than once. After several data samples are generated, these models are then trained independently, and depending on the type of task— i.e. regression or classification—the average or majority of those predictions yield a more accurate estimate. This approach is commonly used to reduce variance within a noisy dataset.

Random forest algorithm

The random forest algorithm is an extension of the bagging method as it utilizes both bagging and feature randomness to create an uncorrelated forest of decision trees. Feature randomness, also known as feature bagging or "the random subspace method"(link resides outside IBM) (PDF, 121 KB), generates a random subset of features, which ensures low correlation among decision trees. This is a key difference between decision trees and random forests. While decision trees consider all the possible feature splits, random forests only select a subset of those features.

If we go back to the "should I surf?" example, the questions that I may ask to determine the prediction may not be as comprehensive as someone else's set of questions. By accounting for all the potential variability in the data, we can reduce the risk of overfitting, bias, and overall variance, resulting in more precise predictions.

How the Random Forest Algorithm Works

The following are the basic steps involved in performing the random forest algorithm:

1. Pick N random records from the dataset.
2. Build a decision tree based on these N records.
3. Choose the number of trees you want in your algorithm and repeat steps 1 and 2.
4. In case of a regression problem, for a new record, each tree in the forest predicts a value for Y (output). The final value can be calculated

by taking the average of all the values predicted by all the trees in forest. Or, in case of a classification problem, each tree in the forest predicts the category to which the new record belongs. Finally, the new record is assigned to the category that wins the majority vote.

HOW DOES IT WORKS?

Random forest algorithms have three main hyperparameters, which need to be set before training. These include node size, the number of trees, and the number of features sampled. From there, the random forest classifier can be used to solve for regression or classification problems.

The random forest algorithm is made up of a collection of decision trees, and each tree in the ensemble is comprised of a data sample drawn from a training set with replacement, called the bootstrap sample. Of that training sample, one-third of it is set aside as test data, known as the out-of-bag (oob) sample, which we'll come back to later. Another instance of randomness is then injected through feature bagging, adding more diversity to the dataset and reducing the correlation among decision trees. Depending on the type of problem, the determination of the prediction will vary. For a regression task, the individual decision trees will be averaged, and for a classification task, a majority vote—i.e. the most frequent categorical variable—will yield the predicted class. Finally, the oob sample is then used for cross-validation, finalizing that prediction.

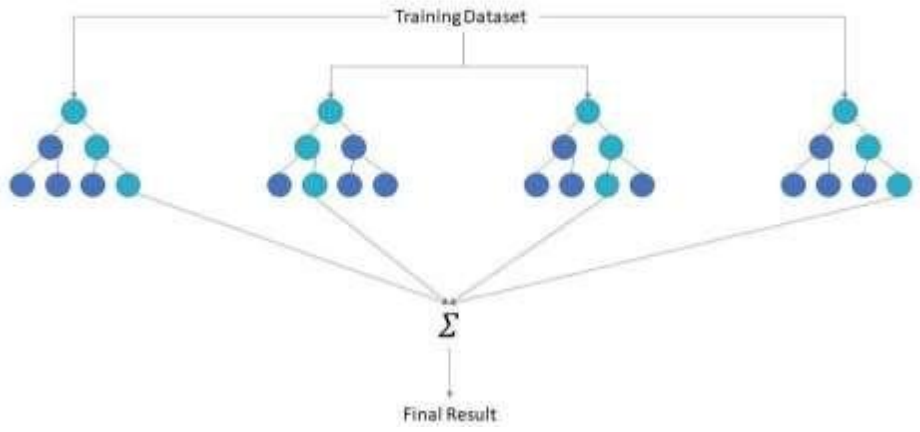


Diagram of Random Forest Classifier

Benefits and challenges of random forest

There are a number of key advantages and challenges that the random forest algorithm presents when used for classification or regression problems. Some of them include:

Key Benefits

- **Reduced risk of overfitting:** Decision trees run the risk of overfitting as they tend to tightly fit all the samples within training data.

However, when there's a robust number of decision trees in a random forest, the classifier won't overfit the model since the averaging of uncorrelated trees lowers the overall variance and prediction error.

- **Provides flexibility:** Since random forest can handle both regression and classification tasks with a high degree of accuracy, it is a popular method among data scientists. Feature bagging also makes the random forest classifier an effective tool for estimating missing values as it maintains accuracy when a portion of the data is missing.
- **Easy to determine feature importance:** Random forest makes it easy to evaluate variable importance, or contribution, to the model. There are a few ways to evaluate feature importance. Gini importance and mean decrease in impurity (MDI) are usually used to measure how much the model's accuracy decreases when a given variable is excluded. However, permutation importance, also known as mean decrease accuracy (MDA), is another importance measure. MDA identifies the average decrease in accuracy by randomly permutating the feature values in oob samples.

Key Challenges

- **Time-consuming process**: Since random forest algorithms can handle large data sets, they can be provide more accurate predictions, but can be slow to process data as they are computing data for each individual decision tree.
- **Requires more resources**: Since random forests process larger data sets, they'll require more resources to store that data.
- **More complex**: The prediction of a single decision tree is easier to interpret when compared to a forest of them.

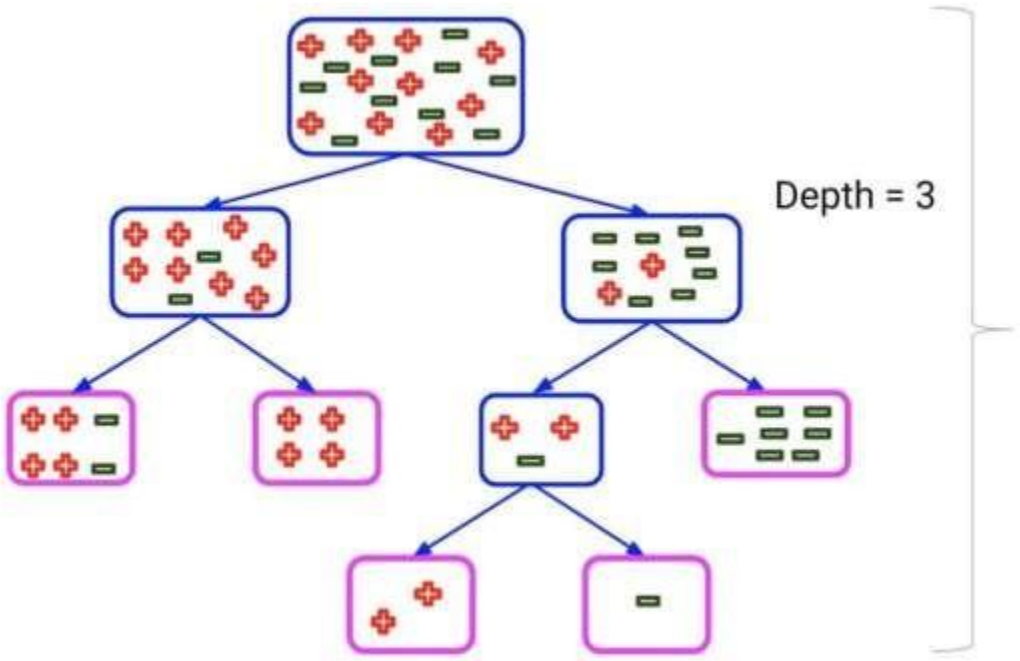
Random Forest Hyperparameters we'll be Looking at:

- max_depth
- min_sample_split
- max_leaf_nodes
- min_samples_leaf
- n_estimators
- max_sample (bootstrap sample)
- max_features

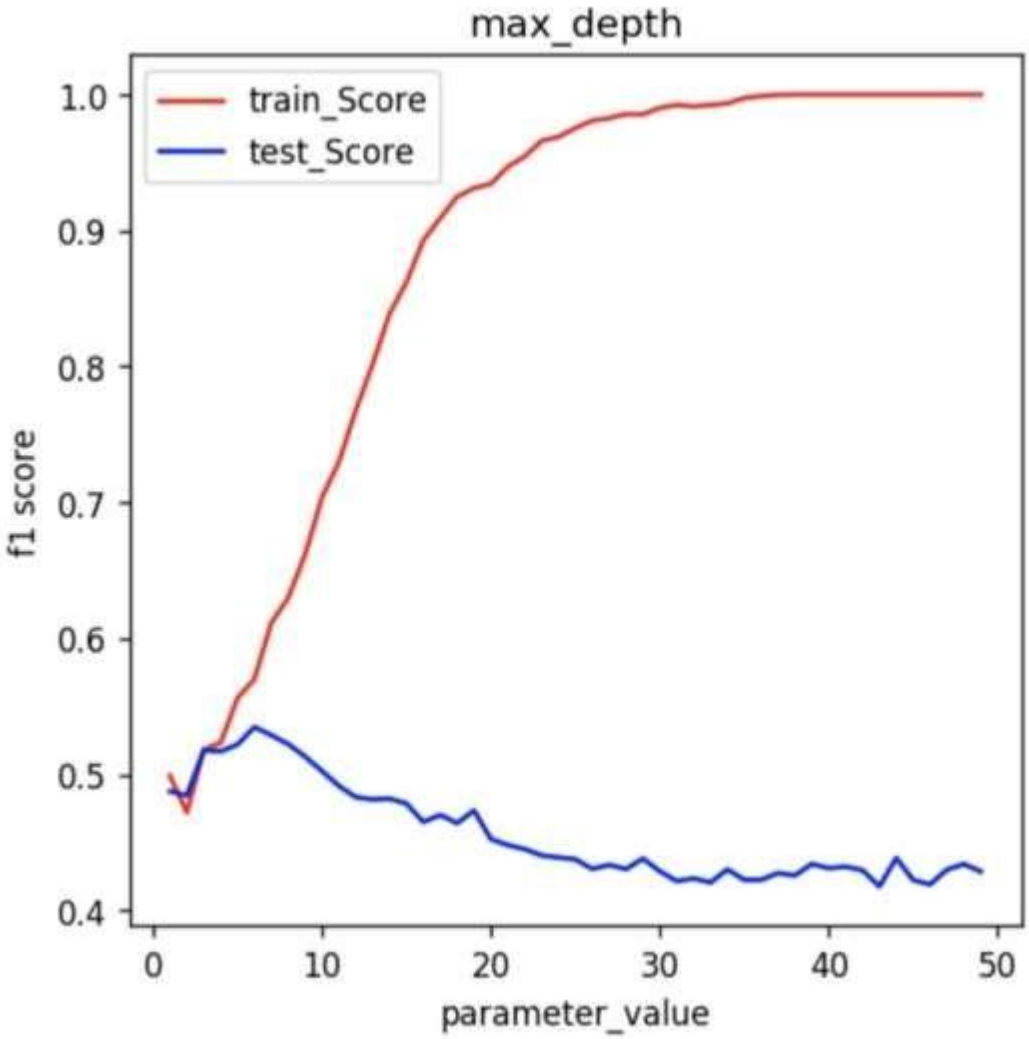
Random Forest Hyperparameter #1: max_depth

Let's discuss the critical *max_depth* hyperparameter first. The *max_depth* of a tree in

Random Forest is defined as the longest path between the root node and the leaf node:



Using the *max_depth* parameter, I can limit up to what depth I want every tree in my random forest to grow.



In this graph, we can clearly see that as the max depth of the decision tree increases, the performance of the model over the training set increases continuously. On the other hand as the *max_depth* value increases, the performance over the test set increases initially but after a certain point, it starts to decrease rapidly.

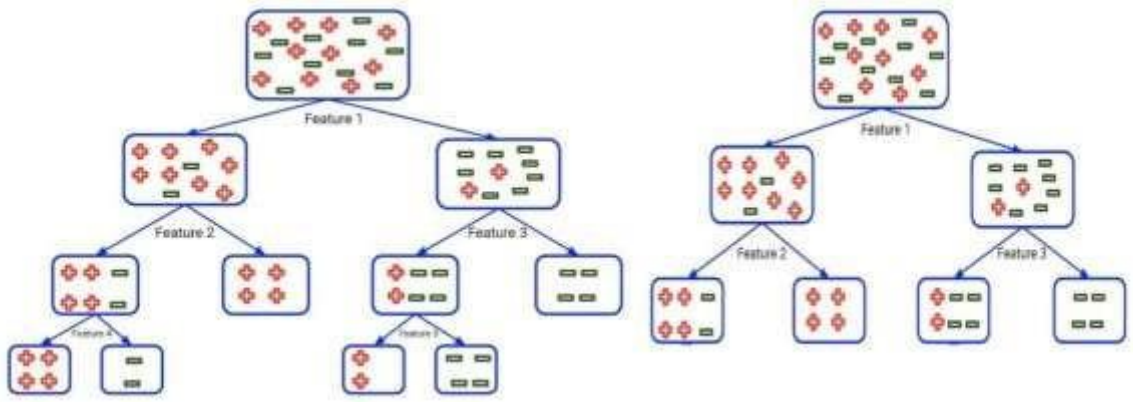
Can you think of a reason for this? The tree starts to overfit the training set and therefore is not able to generalize over the unseen points in the test set.

Among the parameters of a decision tree, *max_depth* works on the macro level by greatly reducing the growth of the Decision Tree.

Random Forest Hyperparameter #2: min_sample_split *min_sample_split* – a parameter that tells the decision tree in a random forest the minimum required number of observations in any given node in order to split it.

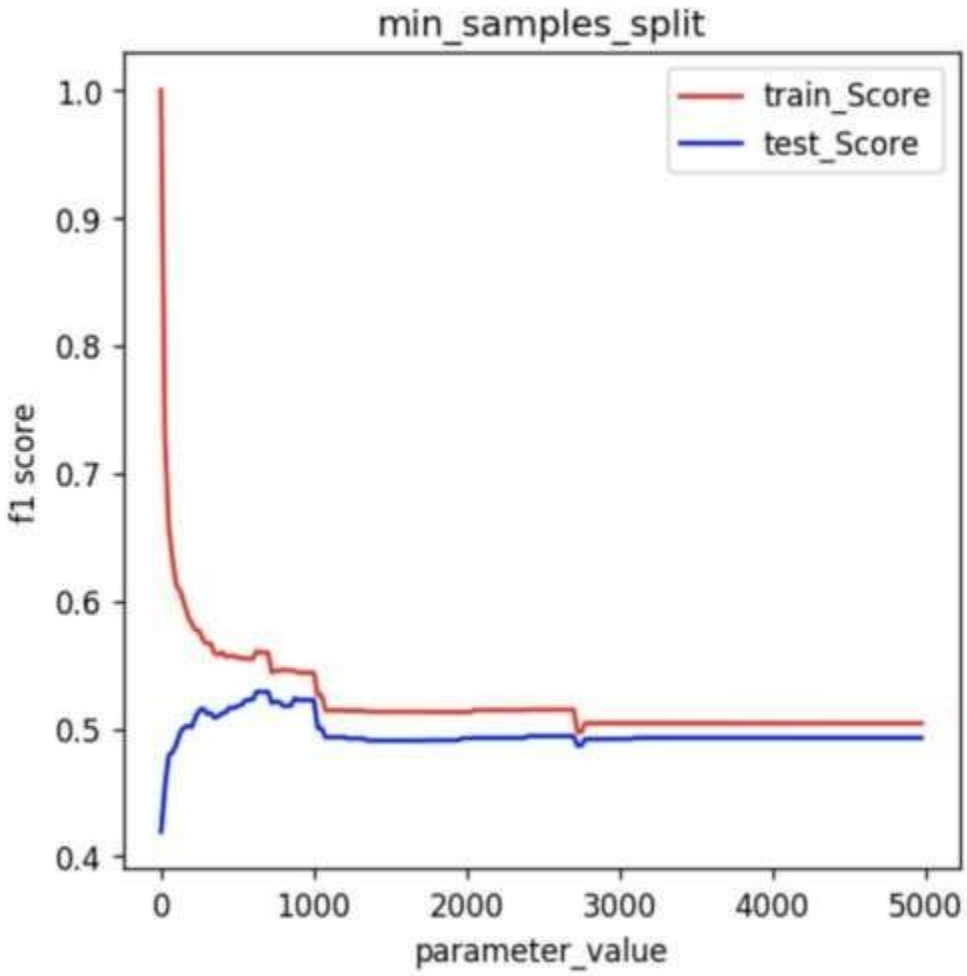
The default value of the minimum_sample_split is assigned to 2. This means that if any terminal node has more than two observations and is not a pure node, we can split it further into subnodes.

Having a default value as 2 poses the issue that a tree often keeps on splitting until the nodes are completely pure. As a result, the tree grows in size and therefore overfits the data.



By increasing the value of the *min_sample_split*, we can reduce the number of splits that happen in the decision tree and therefore prevent the model from overfitting. In the above example, if we increase the *min_sample_split* value from 2 to 6, the tree on the left would then look like the tree on the right.

Now, let's look at the effect of *min_samples_split* on the performance of the model. The graph below is plotted considering that all the other parameters remain the same and only the value of *min_samples_split* is changed:



On increasing the value of the *min_sample_split* hyperparameter, we can clearly see that for the small value of parameters, there is a significant difference between the training score and the test scores. But as the value of the parameter increases, the difference between the train score and the test score decreases.

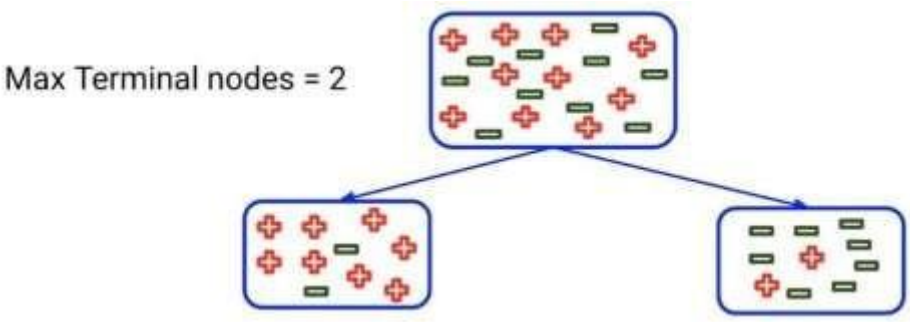
But there's one thing you should keep in mind. *When the parameter value increases too much, there is an overall dip in both the training score and test scores. This is due to the fact that the minimum requirement of splitting a node is so high that there are no significant splits observed. As a result, the random forest starts to underfit.*

You can read more about the concept of overfitting and underfitting here: .

Underfitting vs. Overfitting in Machine Learning

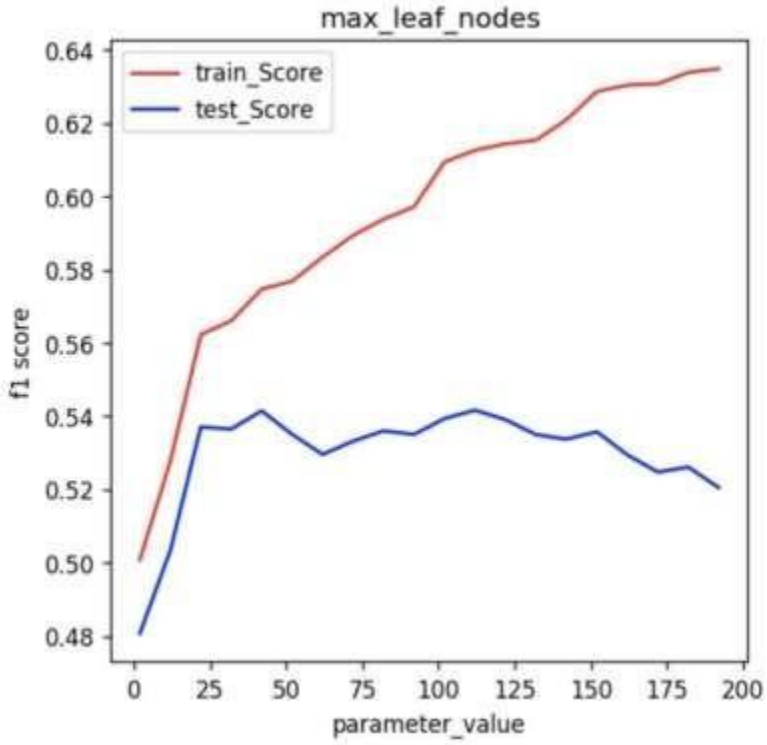
Random Forest Hyperparameter #3: max_terminal_nodes Next, let's move on to another Random Forest hyperparameter called *max_leaf_nodes*. **This hyperparameter sets a condition on the splitting of the nodes in the tree and hence restricts the growth of the tree.** If after splitting we have more terminal nodes than the specified number of terminal nodes, it will stop the splitting and the tree will not grow further.

Let's say we set the maximum terminal nodes as 2 in this case. As there is only one node, it will allow the tree to grow further:



Now, after the first split, you can see that there are 2 nodes here and we have set the maximum terminal nodes as 2. Hence, the tree will terminate here and will not grow further. This is how setting the maximum terminal nodes or *max_leaf_nodes* can help us in preventing overfitting.

Note that if the value of the *max_leaf_nodes* is very small, the random forest is likely to underfit. Let's see how this parameter affects the random forest model's performance:

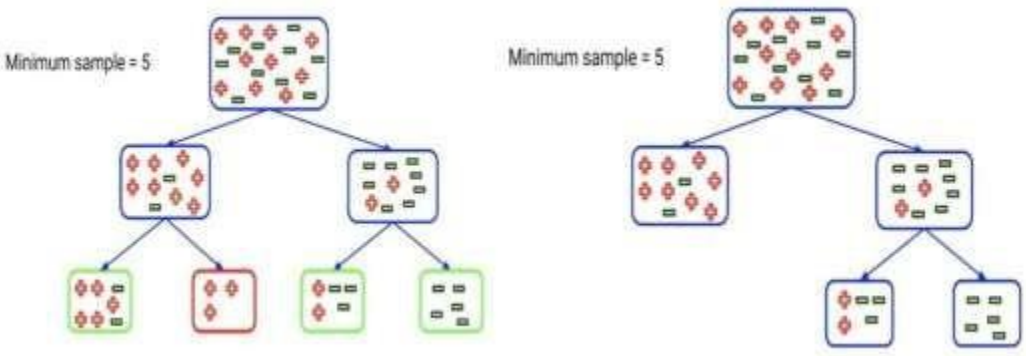


We can see that when the parameter value is very small, the tree is underfitting and as the parameter value increases, the performance of the tree over both test and train increases. According to this plot, the tree starts to overfit as the parameter value goes beyond 25.

Random Forest Hyperparameter #4: min_samples_leaf

Time to shift our focus to *min_sample_leaf*. This Random Forest hyperparameter specifies the minimum number of samples that should be present in the leaf node **after splitting** a node.

Let's understand *min_sample_leaf* using an example. Let's say we have set the minimum samples for a terminal node as 5:

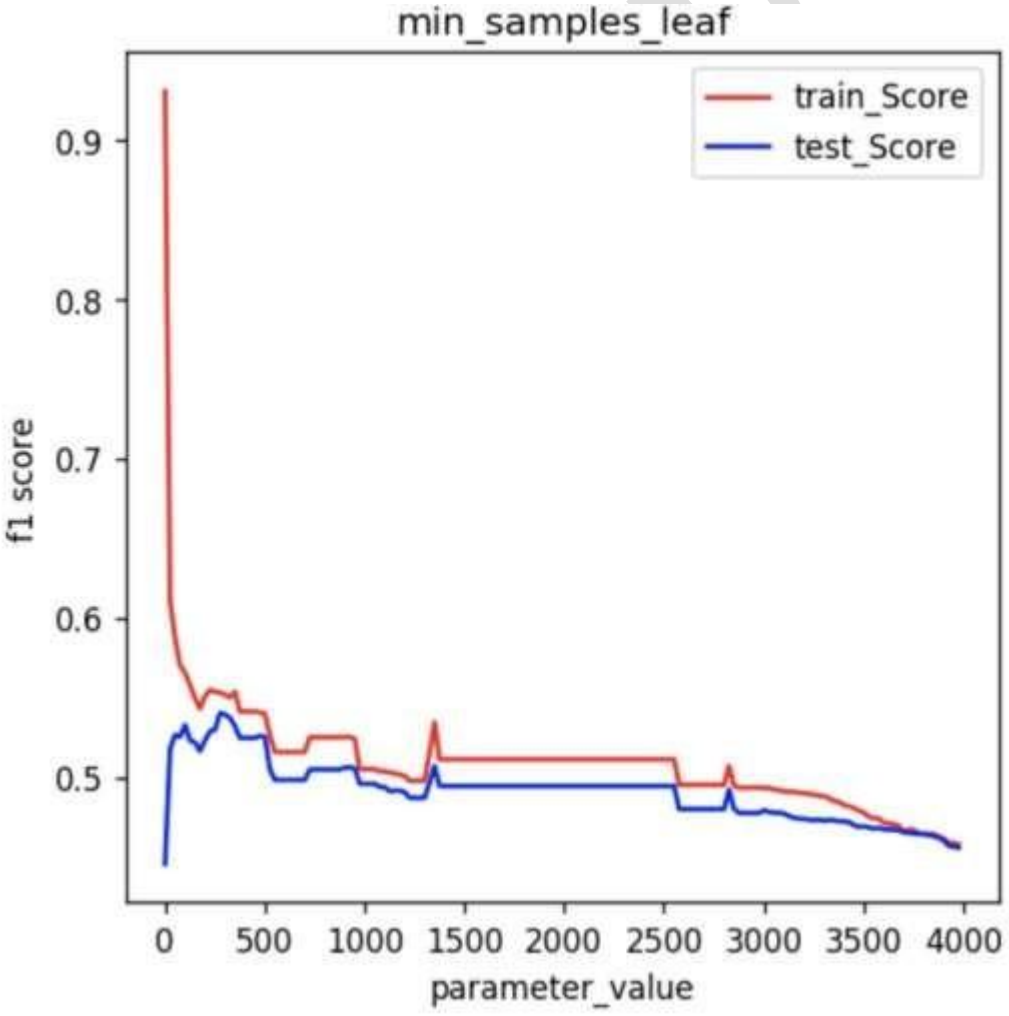


The tree on the left represents an unconstrained tree. Here, the nodes marked with green color satisfy the condition as they have a minimum of 5 samples. Hence, they will be treated as the leaf or terminal nodes.

However, the red node has only 3 samples and hence it will not be considered as the leaf node. Its parent node will become the leaf node. That's why the tree on the right represents the results when we set the minimum samples for the terminal node as 5.

So, we have controlled the growth of the tree by setting a minimum sample criterion for terminal nodes. As you would have guessed, similar to the two hyperparameters mentioned above, this hyperparameter also helps prevent overfitting as the parameter value increases.

If we plot the performance/parameter value plot as before:



We can clearly see that the Random Forest model is overfitting when the parameter value is very low (when parameter value < 100), but the model performance quickly rises up and rectifies the issue of overfitting (100 <

parameter value < 400). But when we keep on increasing the value of the parameter (> 500), the model slowly drifts towards the realm of underfitting.

So far, we have looked at the hyperparameters that are also covered in Decision Trees.

Let's now look at the hyperparameters that are exclusive to Random Forest. Since Random Forest is a collection of decision trees, let's begin with the number of estimators.

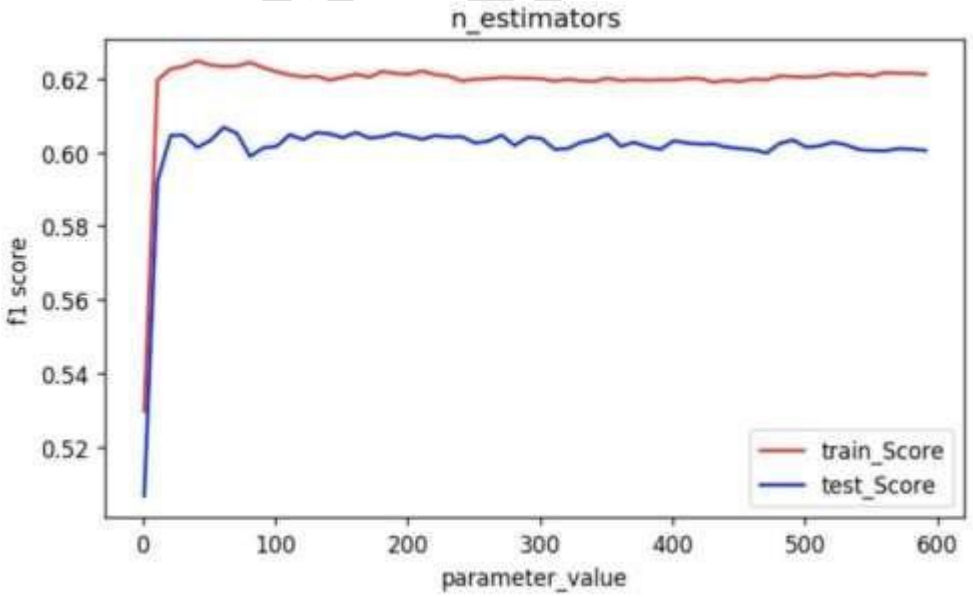
Random Forest Hyperparameter #5: n_estimators

We know that a Random Forest algorithm is nothing but a grouping of trees. But how many trees should we consider? That's a common question fresher data scientists ask.

And it's a valid one!

We might say that more trees should be able to produce a more generalized result, right? But by choosing more number of trees, the time complexity of the Random Forest model also increases.

In this graph, we can clearly see that the performance of the model sharply increases and then stagnates at a certain level:

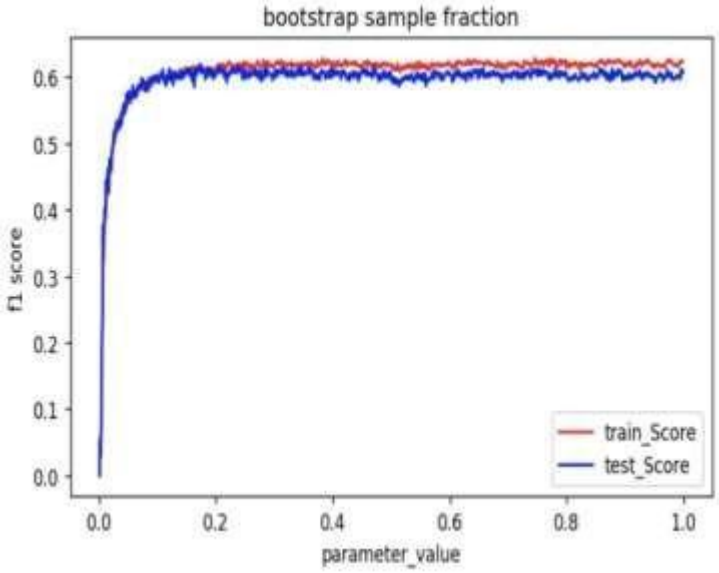


This means that choosing a large number of estimators in a random forest model is not the best idea. Although it will not degrade the model, it can save you the computational complexity and prevent the use of a fire extinguisher on your CPU!

Random Forest Hyperparameter #6: max_samples

The *max_samples* hyperparameter determines what fraction of the original dataset is given to any individual tree. You might be thinking that more data is always better.

Let's try to see if that makes sense.



We can see that the performance of the model rises sharply and then saturates fairly quickly. Can you figure out what the key takeaway from this visualization is?

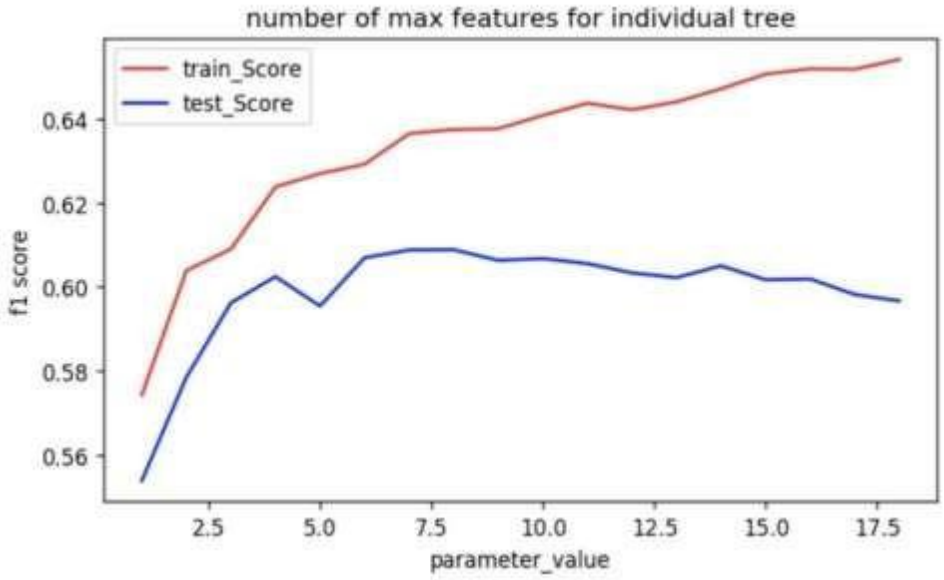
It is not necessary to give each decision tree of the Random Forest the full data. If you would notice, the model performance reaches its max when the data provided is less than 0.2 fraction of the original dataset. That's quite astonishing!

Although this fraction will differ from dataset to dataset, we can allocate a lesser fraction of bootstrapped data to each decision tree. As a result, the training time of the Random Forest model is reduced drastically.

Random Forest Hyperparameter #7: max_features

Finally, we will observe the effect of the *max_features* hyperparameter. This resembles the number of maximum features provided to each tree in a random forest.

We know that random forest chooses some random samples from the features to find the best split. Let's see how varying this parameter can affect our random forest model's performance.



We can see that the performance of the model initially increases as the number of *max_feature* increases. But, after a certain point, the *train_score* keeps on increasing. But the *test_score* saturates and even starts decreasing towards the end, which clearly means that the model starts to overfit.

Ideally, the overall performance of the model is the highest close to 6 value of the max features. It is a good convention to consider the default value of this parameter, which is set to square root of the number of features present in the dataset. The ideal number of max_features generally tend to lie close to this value.,

Random forest applications

The random forest algorithm has been applied across a number of industries, allowing them to make better business decisions. Some use cases include:

- **Finance**: It is a preferred algorithm over others as it reduces time spent on data management and pre-processing tasks. It can be used to evaluate customers with high credit risk, to detect fraud, and option pricing problems.
- **Healthcare:** The random forest algorithm has applications within computational biology (link resides outside IBM) (PDF, 737 KB), allowing doctors to tackle problems such as gene expression classification, biomarker discovery, and sequence annotation. As a result, doctors can make estimates around drug responses to specific medications.

- **E-commerce**: It can be used for recommendation engines for cross-sell purposes