Linear Models

Our last module got little mathematical, and rightly so, for it forms the basis of much what follows next. However, That doesn't mean things are going to get even more complex. No, we'll focus more and more on practical aspects of ML as we move forward and see how the gap is bridged between the math and eventually its application in business.

You'll also notice that when it comes to application of things, its good to know math to understand whats going on at the back end; but its not absolutely necessary. You can always be "not so confident" about the math of things and still implement standard algorithms with much ease and accuracy with what we are going to learn next .

Meaning, you can always come back and have another go at making sense of all the mathematical jargon associated , but it shouldn't become a hurdle in you advancing through the implementation and usage of these algorithms. Persevere!

Lets summarize some relevant bits which we are going to refer to time and again.

There were two kinds of business problems :

1. Regression (with continuous values as target)
2. Classification (with [fixed]categorical values as target)

For Regression :

Predictive Model : $y_i \ = \ f(X_i)$

Cost/Loss : $\sum_{i=1}^{i=n}(y_i \ - \ f(X_i))^2$

For Classification :

Predictive Model : $p_i \ = \ \frac{1}{1+e^{-f(X_i)}}$

Cost/Loss : $-\sum_{i=1}^{i=n}[y_i * log(p_i) + (1-y_i) * log(1-p_i)]$

In both of these cases $f(X_i)$ can be any generic function, which we'll be referring to as Algorithms. In case of Linear Models , $f(X_i)$ is linear combination of input variables of the problem (for both regression and classification)

Formally; for linear models :

$f(X_i) \ = \ \beta_0 \ + \ \beta_1 * x_{1i} \ + \ \beta_2 * x_{2i} \ + \ \beta_3 * x_{3i} . . . . .$

It seems our discussion on linear models should be finished here and we should move on to other parts of the course already. No, not really. There are many unanswered questions here before we start to build practical, industry level linear models. Here are those questions which we'll address one by one

1. All the discussions on theoretical aspects have made very convenient assumption that all the inputs for predicting the outcomes are going to be numeric, which is definitely not the case.

For example , if we are trying to predict sales of different shops; part of a retail chain, which city they are located in can be a very important feature to consider. However in our discussion so far, we have not considered how we convert this inherently categorical information into numbers and use it in our algorithms.

2. How do I know, how good are my predictions. Nobody in industry will accept my solution because I developed it, they need to know how it will tentatively perform before it makes into production
3. It seems that, if I pass data on 100 inputs/variables to this algorithm to predict the outcome of interest, it will give coefficient for all the inputs. Meaning, all the inputs are going to be part of the predictive model or in other words; all of them will have some impact on my predictions; irrespective of some of them being junk inputs. Our algorithm should contain some way of either completely removing them from the model or suppress their impact on our predictions.

First Question : Handling Categorical Variable

We'll start with the first question . Categorical variables are converted to dummies is the long and short of it. Lets figure out why?

In the context of numeric variables we can say that if the variable changed by some Δx amount then, our prediction will change by some constant multiplied by Δx amount. However this concept of change by Δx amount , just doesn't exist in context of categorical variables. Consider a case where we were trying to build a predictive model for how much time people take to run a 100 Meter dash, given their age and what kind of terrain they run on [hilly/flat]. You cant really say that terrain changed by `flat-hill`. All that you can say that people running on a hilly terrain , on an average will take some more time, assuming effect of age remains same across the population. We can depict this difference by using two separate predictive equations for both terrains

$$Runtime_{hills} = 10 + 0.5 * Age + 5$$
$$Runtime_{flats} = 10 + 0.5 * Age$$

Note : constant used here are indicative values

You can see; these two equations mean that if two people of same age run on two different terrains , person running on hills will take 5 seconds more. Does that mean if I have 100 categories in my categorical variable, I'll have to make 100 different models ? Not really , we can easily combine them like this :

$$Runtime_{hills} = 10 + 0.5 * Age + 5 * Terrain_{hill}$$

This one equation represents both the above seen separate equations. Variable $Terrain_{hill}$ takes value 1, when terrain is `hill` and value 0 when terrain is `flat`. This is called a dummy/flag variable. It is also known as one hot encoded representation of categorical data. Notice that we had two categories in our categorical variable `Terrain` , But we need only one dummy variable to represent it numerically . In general if our categorical variable has `n` categories , we need only `n-1` dummies. Theoretically it doesn't matter which category we ignore while creating dummies for rest.

One last thing that you need to keep in mind is that, since categories have an average impact; for their estimated average impact from the data to be consistent/reliable; they need to have backing of good number of observations. Essentially we should ignore categories which have too few obs in the data. Is there a magic number which we should consider as minimum required number of observations ? No, there isn't . A good sane choice will do, you can even experiment with different numbers.

Second Question : Validation

We'll start this discussion on this question , with another one , `why are we building this predictive model ?`

Simple answer is that we want to make use of that model to make prediction on future data, ideally we'd like to know the performance before data. But there is no way to get that future data. We'll have to make do with whatever training data we have been given.

It doesnt make sense to check performance of the model on the same data it was trained on, Since the model has already seen the outcome , it ofcourse will perform on the same data as well as possible. That can not be taken as measure of its performance on unseen data. There are two ways , both of which have their pros/cons.

Breaking Training Data into Two Parts:

This is one of the simpler ways , but not without its flaws. You break your training data randomly in two parts , build the model on one and test its performance on another. Generally the training data is broken into 80:20, larger part to be used for training and smaller one to test performance. But again that isn't a magic ratio. Idea is to keep a small sample separate from the training data, but not too small (so that it contains similar patterns from the overall data) but not too big (so as to not have different patterns from the overall data).

Pros : Its quick and easy way to validate model

Cons : We don't have good idea about , what can be optimal ratio which is balance between not too big or too small . This random sample might be a niche part of the data having very different pattern from the rest of the data and in that case , measured performance will not be a good representative of real performance of the model

Cross-Validation :

Instead of breaking it into two parts , we break it into `K` parts. A good value of K is in the range 10-15. You'll have a better understanding of `why` , once we are done with this discussion .

Problem with simply breaking data into two parts was that we cant ensure that smaller validation set really resembles the overall data. Performance results on this will vary from the real performance. To counter that we break data into K parts. And build K models , every time leaving one of the parts as validation sample. This gives us out of sample performance measures for these K parts. Instead of using one of them as representative tentative performance. We take the average of these performance measures . Upon averaging , variations will be canceled out and we'll have more representative measure of performance. We can also look at variance of this performance to asses how stable it is across data.

Taking K as 10 means, at a time 10% of the data is not involved in the modeling process. Its fair to say 90% of the data will still have same pattern as the overall data. This number goes up as we break our data into more parts by taking higher value of K, but that has its down side of increase number of models we built (increasing the time taken)

Pros : Gives better measure of performance along with variance as stability measure .

Cons : Its time/resource taking, might as well be infeasible for very large datasets.

Lets build a simple linear regression model

We'll resume our discussion on the third and final question after this section . In this section we are going to revisit data preparation with pandas and learn about sci-kit learns interface for building machine learning models

Problem Statement : We have been given data for people applying for a loan to a peer-to-peer lending firm. The data contains details of loan application and eventually how much interest rate was offered to people by the platform. Our solution needs to be able to predict the interest rate which will be offered to people , given their application detail as inputs. Lets look at the training data given to us

```
# import for processing data
import pandas as pd
import numpy as np
# imports for suppressing warnings
import warnings
warnings.filterwarnings('ignore')
```

```
# provide complete path for the file which contains your data
# r at the beginning is used to ensure that path is considerd as raw string and
# you dont get unicode error because of special characters combined with \ or /

file=r'/Users/lalitsachan/Dropbox/0.0 Data/loan_data_train.csv'

ld_train=pd.read_csv(file)
```

```
ld_train.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2200 entries, 0 to 2199
Data columns (total 15 columns):
ID                                2199 non-null float64
Amount.Requested                  2199 non-null object
Amount.Funded.By.Investors        2199 non-null object
Interest.Rate                     2200 non-null object
```

```
Loan.Length                       2199 non-null object
Loan.Purpose                      2199 non-null object
Debt.To.Income.Ratio              2199 non-null object
State                             2199 non-null object
Home.Ownership                    2199 non-null object
Monthly.Income                    2197 non-null float64
FICO.Range                        2200 non-null object
Open.CREDIT.Lines                 2196 non-null object
Revolving.CREDIT.Balance          2197 non-null object
Inquiries.in.the.Last.6.Months    2197 non-null float64
Employment.Length                 2131 non-null object
dtypes: float64(3), object(12)
memory usage: 257.9+ KB
```

Lets comment on each of these variables one by one , as to what we are going to do before we start building our model

ID : It doesn't make sense to include unique identifiers of the observation (ID vars) as input. We'll drop this column

Amount.Requested , Open.CREDIT.Lines, Revolving.CREDIT.Balance : Ideally these should have been numeric columns, but if you look at the type assigned , it is object type. They must have come as object type because of some odd strings in the data at one or more places . We'll convert them to numeric type.

Amount.Funded.By.Investors : This information, although present in the data, will not come with loan application. If we want to build a model for predicting Interest.Rate using loan application characteristics , then we can not include this information in our model. We'll drop this column

Interest.Rate, Debt.To.Income.Ratio: These come as object type again because of the % symbol contained in it. We'll first remove the % sign and then convert it to numeric type

```
ld_train['Interest.Rate'].head()
```

```
0    18.49%
1    17.27%
2    14.33%
3    16.29%
4    12.23%
Name: Interest.Rate, dtype: object
```

```
ld_train['Debt.To.Income.Ratio'].head()
```

```
0    27.56%
1    13.39%
2     3.50%
3    19.62%
4    23.79%
Name: Debt.To.Income.Ratio, dtype: object
```

State , Home.Ownership, Loan.Length, Loan.Purpose : We'll create dummies , ignoring categories with too few occurrences

```
ld_train['State'].value_counts(dropna=False).head()
# only partial results are shown. to see the full results , remove .head()
```

```
CA    376
NY    231
FL    149
TX    146
PA     88
Name: State, dtype: int64
```

```
ld_train['Home.Ownership'].value_counts(dropna=False).head()
```

```
MORTGAGE    1018
RENT         999
OWN          177
OTHER          4
NONE           1
Name: Home.Ownership, dtype: int64
```

```
ld_train['Loan.Length'].value_counts(dropna=False)
```

```
36 months    1722
60 months     476
.               1
NaN             1
Name: Loan.Length, dtype: int64
```

```
ld_train['Loan.Purpose'].value_counts(dropna=False).head()
# only partial results are shown. to see the full results , remove .head()
```

```
debt_consolidation    1147
credit_card            394
other                  174
home_improvement       135
major_purchase          84
Name: Loan.Purpose, dtype: int64
```

Monthly.Income , Inquiries.in.the.Last.6.Months : Leave as is

FICO.Range: This comes as object type because the value written as numeric ranges in the data. As such, we can convert this to dummies, but we'll not be using information contained in order of the values if we convert them to dummies . We'll instead take average of the given range using string processing .

```
ld_train['FICO.Range'].value_counts().head()
```

```
670-674    151
675-679    144
680-684    141
695-699    138
665-669    129
Name: FICO.Range, dtype: int64
```

Employment.Length: This takes type object; because it takes numeric values written in words. We can again chose to work with it like a categorical variable, but then we'll endup losing information on the order of values .

```
ld_train['Employment.Length'].value_counts(dropna=False)
```

```
10+ years    575
< 1 year     229
2 years      217
3 years      203
5 years      181
4 years      162
1 year       159
6 years      134
7 years      109
8 years       95
NaN           69
9 years       66
.              1
Name: Employment.Length, dtype: int64
```

Lets begin with these operations that we have decided :

```
# removing ID and Amount.Funded.By.Investors
ld_train.drop(['ID','Amount.Funded.By.Investors'],axis=1,inplace=True)
```

```
# Removing % signs from two columns
for col in ['Interest.Rate','Debt.To.Income.Ratio']:
    ld_train[col]=ld_train[col].str.replace("%","")
```

```
# converting columns to numeric with pandas

for col in ['Amount.Requested', 'Interest.Rate','Debt.To.Income.Ratio',
            'Open.CREDIT.Lines','Revolving.CREDIT.Balance']:
    ld_train[col]=pd.to_numeric(ld_train[col],errors='coerce')
```

```
# Processing FICO.Range
k=ld_train['FICO.Range'].str.split("-",expand=True).astype(float)

ld_train['fico']=0.5*(k[0]+k[1])

del ld_train['FICO.Range']
```

```
# Processing Employment.Length

ld_train['Employment.Length']=ld_train['Employment.Length'].str.replace('years','')
ld_train['Employment.Length']=ld_train['Employment.Length'].str.replace('year','')
ld_train['Employment.Length']=np.where(ld_train['Employment.Length'].str[0]=='<',0,
                                       ld_train['Employment.Length'])
ld_train['Employment.Length']=np.where(ld_train['Employment.Length'].str[:2]=='10',10,
                                       ld_train['Employment.Length'])
ld_train['Employment.Length']=pd.to_numeric(ld_train['Employment.Length'],errors='coerce')

```

```
# Creating dummies with frequency cutoff

cat_col=['State' , 'Home.Ownership', 'Loan.Length', 'Loan.Purpose']
# this will be done for each of the columns in cat_col
for col in cat_col :
    # calculate frequency of categories in the columns
    k=ld_train[col].value_counts(dropna=False)
    # ignoring categories with too low frequencies and then selecting n-1 to create dummies for
    cats=k.index[k>50][:-1]
    # creating dummies for remaining categories
    for cat in cats:
        # creating name of the dummy column coresponding to the category
        name=col+'_'+cat
        # adding the column to data
        ld_train[name]=(ld_train[col]==cat).astype(int)
    # removing the original column once we are done creating dummies for it
    del ld_train[col]


```

```
ld_train.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2200 entries, 0 to 2199
Data columns (total 31 columns):
Amount.Requested                    2195 non-null float64
Interest.Rate                       2200 non-null float64
Debt.To.Income.Ratio                2199 non-null float64
Monthly.Income                      2197 non-null float64
```

```
Open.CREDIT.Lines                    2193 non-null float64
Revolving.CREDIT.Balance             2195 non-null float64
Inquiries.in.the.Last.6.Months       2197 non-null float64
Employment.Length                    2130 non-null float64
fico                                 2200 non-null float64
State_CA                             2200 non-null int64
State_NY                             2200 non-null int64
State_FL                             2200 non-null int64
State_TX                             2200 non-null int64
State_PA                             2200 non-null int64
State_IL                             2200 non-null int64
State_GA                             2200 non-null int64
State_NJ                             2200 non-null int64
State_VA                             2200 non-null int64
State_MA                             2200 non-null int64
State_NC                             2200 non-null int64
State_OH                             2200 non-null int64
State_MD                             2200 non-null int64
State_CO                             2200 non-null int64
Home.Ownership_MORTGAGE              2200 non-null int64
Home.Ownership_RENT                  2200 non-null int64
Loan.Length_36 months                2200 non-null int64
Loan.Purpose_debt_consolidation      2200 non-null int64
Loan.Purpose_credit_card             2200 non-null int64
Loan.Purpose_other                   2200 non-null int64
Loan.Purpose_home_improvement        2200 non-null int64
Loan.Purpose_major_purchase          2200 non-null int64
dtypes: float64(9), int64(22)
memory usage: 532.9 KB
```

All of the columns in the data are now numeric. Just one more thing to take care of before we start with modeling process. We need to make sure that there are no missing values in the data. if there are , then they need to be replaced .

```
# checking for missing values in the data

ld_train.isnull().sum()
```

```
Amount.Requested                     5
Interest.Rate                        0
Debt.To.Income.Ratio                 1
Monthly.Income                       3
Open.CREDIT.Lines                    7
Revolving.CREDIT.Balance             5
Inquiries.in.the.Last.6.Months       3
Employment.Length                   70
```

```
fico                                0
State_CA                            0
State_NY                            0
State_FL                            0
State_TX                            0
State_PA                            0
State_IL                            0
State_GA                            0
State_NJ                            0
State_VA                            0
State_MA                            0
State_NC                            0
State_OH                            0
State_MD                            0
State_CO                            0
Home.Ownership_MORTGAGE             0
Home.Ownership_RENT                 0
Loan.Length_36 months               0
Loan.Purpose_debt_consolidation     0
Loan.Purpose_credit_card            0
Loan.Purpose_other                  0
Loan.Purpose_home_improvement       0
Loan.Purpose_major_purchase         0
dtype: int64
```

```
# imputing missing values with averages of the columns

for col in ld_train.columns:
    if ld_train[col].isnull().sum()>0:
        ld_train.loc[ld_train[col].isnull(),col]=ld_train[col].mean()

```

First we'll use the first method of validation , where we break data into two parts before start building the model

```
# breaking data into two parts

from sklearn.model_selection import train_test_split

t1,t2=train_test_split(ld_train,test_size=0.2,random_state=123)
# test_size=0.2, means the data is being split into two parts in the 80:20 ratio.
# t1 will contain 80%, and t2 will get 20% of the obs.
# random_state=123, simply makes the random process reproducible
```

we need to separate ppredictors (input/xvars) and target before we pass them to scikit-learn functions for building our linear regression model

```
x_train=t1.drop('Interest.Rate',axis=1)
y_train=t1['Interest.Rate']
x_test=t2.drop('Interest.Rate',axis=1)
y_test=t2['Interest.Rate']
```

Lets build our first linear regression model

```
# import the function for Linear Regression

from sklearn.linear_model import LinearRegression

lr=LinearRegression()
# fit function , builds the model ( parameter estimation etc)
lr.fit(x_train,y_train)
```

```
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None,
         normalize=False)
```

Lets looks at estimated coefficients . intercept is same as β_0

```
lr.intercept_
```

```
76.32100980688705
```

```
list(zip(x_train.columns,lr.coef_))
```

```
[('Amount.Requested', 0.00016206823832747567),
 ('Debt.To.Income.Ratio', -0.005217635523226176),
 ('Monthly.Income', -4.033914729649962e-05),
 ('Open.CREDIT.Lines', -0.03015894666977517),
 ('Revolving.CREDIT.Balance', -1.7860242337434248e-06),
 ('Inquiries.in.the.Last.6.Months', 0.32786067084992604),
 ('Employment.Length', 0.02325230583339998),
```

```
 ('fico', -0.08716732836779102),
 ('State_CA', -0.16231739562106098),
 ('State_NY', -0.14426278883807817),
 ('State_FL', -0.11716306311499997),
 ('State_TX', 0.4481165264861161),
 ('State_PA', -0.9332596674212796),
 ('State_IL', -0.4048740473139449),
 ('State_GA', -0.33202157322249337),
 ('State_NJ', -0.49634957660360035),
 ('State_VA', -0.13349751801583823),
 ('State_MA', -0.1634714204731154),
 ('State_NC', -0.47136779712009375),
 ('State_OH', -0.40429922213664504),
 ('State_MD', -0.1292878863756837),
 ('State_CO', 0.10071894446013128),
 ('Home.Ownership_MORTGAGE', -0.5636395222756556),
 ('Home.Ownership_RENT', -0.27130802518538744),
 ('Loan.Length_36 months', -3.1821676438146373),
 ('Loan.Purpose_debt_consolidation', -0.482384755055442),
 ('Loan.Purpose_credit_card', -0.5726731705822421),
 ('Loan.Purpose_other', 0.35159491851815755),
 ('Loan.Purpose_home_improvement', -0.4952547468027438),
 ('Loan.Purpose_major_purchase', -0.2391664596860732)]
```

Lets make predictions for performance check.

```
predicted_values=lr.predict(x_test)
```

```
from sklearn.metrics import mean_absolute_error
```

```
mean_absolute_error(predicted_values,y_test)
```

```
1.6531699740032333
```

This means that, we can assume that tentatively our model, will be off by 1.65 units on an average; while predicting interest rates on the basis of loan application. Now lets see, how we can do cross-validation with sklearn tools . Keep in mind that we don't need to break our data in this process. sklearn function will take care of that internally . All that we need to do is to separate target and predictors .

```
x_train=ld_train.drop('Interest.Rate',axis=1)
y_train=ld_train['Interest.Rate']
```

```
from sklearn.model_selection import cross_val_score
```

```
errors = np.abs(cross_val_score(lr,x_train,y_train,cv=10,scoring='neg_mean_absolute_error'))
# cv=10 , means 10 fold cross validation
# Regarding scoring functions, the general theme in scikit learn is , higher the better
# to remain consistent with the same , instead of mean_absolute_error, available function for regression is neg_mean_absolute_error
# we can always wrap that and take positve values [ with np.abs ]
```

```
errors
```

```
array([1.72994607, 1.70800508, 1.75246664, 1.63094103, 1.43860407,
       1.63204347, 1.42609575, 1.56465166, 1.53020947, 1.63953272])
```

```
avg_error=errors.mean()
```

```
error_std=np.std(errors)
```

```
avg_error,error_std
```

```
(1.6052495976597005, 0.10838823631170523)
```

Making Predictions with Real Test Data

What if we were given a separate test file where we eventually had to make prediction on and then submit (like projects). Biggest challenge will be to ensure that , test data eventually needs to have same columns as transformed training data on which we built the model.

Note : you can not check performance on this test data, if there is no response column given .

There are two ways to do so :

1. Combine training and test from the very beginning and then separate once the data prep is done. Build model on train and make predictions on test
2. Build a data-prep pipeline which gives same results for both train and test (We'll learn about this in later modules)

In here we'll give you a quick summary of the first method .

```
#### Combining two data sets

test_file=r'/Users/lalitsachan/Dropbox/0.0 Data/loan_data_test.csv'

ld_test=pd.read_csv(test_file)
```

```
# add an identifier column to both files so that they can be separated later on

ld_test['data']='test'
ld_train['data']='train'

# combine them
ld_all=pd.concat([ld_train,ld_test],axis=0)

# carry out the same data prep steps on ld_all as we did for ld_train

#~~~~~~~~~~ data prep on the ld_all~~~~~~~~~~~~~~~

# make sure that you end up making dummies for column 'data'
# Now separate them

ld_train=ld_all[ld_all['data']=='train']
ld_test=ld_all[ld_all['data']=='test']

ld_test.drop(['data','Interest.Rate'],1,inplace=True)
del ld_train['data']
del ld_all
```

Now you can build model on `ld_train`, use the same model to make prediction on `ld_test` and make submission. Since both the datasets have gone through the same data prep process , they'll have same columns in them

Third Question : How to suppress effect of junk vars

Lets try to understand interpretation of βs .For a model which is written as :

$$f(X_i) = \beta_0 + \beta_1 * x_{1i} + \beta_2 * x_{2i} + \beta_3 * x_{3i} \ldots\ldots$$

Lets pick x_2 to comment on. The coefficient β_2, associated with x_2, simply means that if x_2 changes by 1 unit, my prediction will change by β_2 units. if β_2 is +ve , then my prediction will increase as x_2 increases; if β_2 is -ve, my prediction will decrease when x_2 increases [pointing to negative linear correlation].

But if x_2 was a junk variable, it should not have any impact on my prediction. In order for that to happen, the coefficient associated with it should be zero or very close to zero.

what happens when we add a random variable to our model

Assuming you have a linear regression model, for easy notation consider first one, then two variables. This generalizes to two sets of models.
The first model is

$$I: y_i = \beta_0 + \beta_1 x_{1i} + \epsilon_i$$

the second model is
$$II: y_i = \beta_0 + \beta_1 x_{1i} + \beta_2 x_{2i} + \epsilon_i$$

This is solved by minimizing sum of squared residuals, for model one we want to minimize
$\mathrm{SSR}_1 = \sum_i (y_i - \beta_0 - \beta_1 x_{1i})^2$
and for model two we want to minimize

$$\mathrm{SSR}_2 = \sum_i (y_i - \beta_0 - \beta_1 x_{1i} - \beta_2 x_{2i})^2.$$

Lets say we have found the correct estimators for model 1, then you can obtain that exact same residual sum squares in model two by choosing the same values for β_0, β_1 and letting $\beta_2 = 0$. Now you can find, possibly, a lower sum squares residual by searching for a better value for β_2.

To summarize, the models are nested, in the sense that everything we can model with model 1 can be matched by model two, model two is more general than model 1. So, in the optimization, we have larger freedom with model two; so can always find a better solution.

This has really nothing to do with statistics but is a general fact about optimization. It extrapolates to following conclusion : If we add any variable in our model/data [junk or not junk]; cost function will always decrease . However for junk variables, it will go down by a small amount, and for good variables [which really do impact our target], it goes down by a large amount.

Regularisation

Problem with having junk vars a coefficient is that , their effect on the model doesn't remain consistent . Having lot of junk vars might lead to a model which performs entirely differently on training data and eventual validation/test data. That defies the purpose of building the model in the first place . This situation where our prediction model performs very well on the training data but not so well on test/validation set; is called `Overfitting`. Ways to reduce this problem and make our model more generalizable is called `Regularisation` .

Many at times this is achieved by modifying our cost/loss formulation such that it reduces the impact of junk vars; and in general reduces impact of overfitting by ensuring that our predictive model extracts most generic patterns from the data instead of simply memorizing training data.

We want to reduce impact of junk vars , that can be simply achieved by making their coefficient as close to zero as possible. We can achieve that by adding a penalty to the cost function on the size of βs . Lets understand how that works . Consider there are two variables x_{good} and x_{bad}. Both of them contribute to decrease in our traditional loss $L(\beta)$. x_{good} however results in higher decrease in comparison to x_{bad}

Lets say, contribution of x_{good} towards decrease of $L(\beta)$ is 10, where as for x_{bad}, it is 0.5 . Now we are going to add penalty to our loss formulation on the size of the parameters . Our new loss function will look like this :

$$L' = \text{Traditional Loss} + Penalty$$

this penalty can be anything as long as it serves the purpose to make our model more generalizable. We'll discuss some popular formulations for the same. But those by no means are the only ones which you are theoretically limited to use.

consider this one, known as L2 penalty [alson known as `Ridge` regression in context of linear regression models] :

$$L' = L(\beta) + \lambda * \beta^2$$

We'll expand on the role of λ here in a bit , for now lets consider that to be 1, and lets also consider that; to start with coefficients for x_{good} and x_{bad}, both are 2 . Notice that penalty is always positive, meaning this will increase the loss function for all the vars, higher the parameter size [absolute value] , higher the increase in the loss. Now in the light of new loss formulation, decrease due to x_{good} will be = (10 - 2X2 = 6); and for x_{bad} it'll be = (0.5 - 2X2 = -3.5) .

Clearly this new loss formulation does not decrease because of x_{bad}, during optimisation of loss, coefficient for x_{bad} will be moved to close to zero until decrease because of it becomes positive.

There are many such penalties we'll come across in our course discussions . They'll mainly be variations of the two, namely; L1 and L2 penalty.

cost with L1 penalty [also known as `Lasso`]: $L(\beta) + \lambda * |\beta|$
cost with L2 penalty : $L(\beta) + \lambda * \beta^2$

There are one basic impact of using either of the penalties to the cost function

L1 leads to model reduction but L2 doesnt

Lets consider case of having a single parameter β, same results get extrapolated to higher number of parameters also.

consider cost function with L2 penalty for linear regression (in matrix format as discussed in the earlier module) :

$$= (Y - X\beta)^T(Y - X\beta) + \lambda * \beta^2$$

We'll calculate the gradient and equate it to zero to determine our paramter value

$$\nabla L = 0$$
$$=> -2 * X^T(Y - X\beta) + 2\lambda * \beta = 0$$
$$=> \beta = \frac{X^TY}{X^TX + \lambda}$$

you can see that , by using higher and higher value of λ, you can make parameter to be very close to zero, but there is no way to make it absolutely zero.

Lets see what happens in case of using L1 penalty , here is the cost function with L1 penalty :

$$= (Y - X\beta)^T(Y - X\beta) + \lambda * |\beta|$$

Lets assume that β>0 for this discussion , you can do the exercise for -ve β as well and reach to the same conclusion. For +ve , cost function is :

$$= (Y - X\beta)^T(Y - X\beta) + \lambda * \beta$$

We'll equate the gradient to zero here also and lets see what happens

$$\nabla L = 0$$
$$=> -2 * X^T(Y - X\beta) + \lambda = 0$$
$$=> \beta = \frac{2X^TY - \lambda}{X^TX}$$

You can see that in this case , there does exist some value of λ for which parameter estimate can become 0

conclusiong :

- Both L1 and L2 penalties reduce impact of junk vars on the model
- L1 penalty can result in some coefficient being exactly zero , thus model reduction
- Since both of them have penalty on the parameter size which is dependent on scale of variables , its advised that we should standardize the data if variables are on different scale

Chosing best value of λ with cross validaton

if we take λ=0, it simply leads to zero penalty and our traditional cost/loss formulation. If we take $\lambda \to \infty$, all βs will have to be zero, leading to complete reduction in the model (or no model). There is no mathematical formula for best value of λ, its different for different datasets. What we can do is , to try out different values of lambda and see its cross validated performance , and choose the one for which cross validated performance is best. Lets see how to do that [extending the example taken earlier] with sklearn functions

```
from sklearn.linear_model import Ridge,Lasso
from sklearn.model_selection import GridSearchCV
```

```
# we are going to try out values from 1 to 100
# what we have referring to as lambda is named alpha in the
# sklearn implementation
# we'll pass this dictionary to GridSearchCV function
lambdas=np.linspace(1,100,200)
params={'alpha':lambdas}
```

```
# this is the model for which we are tryin to estimate best value of
lambda
model=Ridge(fit_intercept=True)
```

```
# this function will be trying out all the values of lambdas
# passed to it and record cross-validate performance
# using a custom function we'll extract the results
grid_search=GridSearchCV(model,param_grid=params,cv=10,scoring='neg_mean_absolute_error')
```

```
grid_search.fit(x_train,y_train)
```

```
GridSearchCV(cv=10, error_score='raise-deprecating',
       estimator=Ridge(alpha=1.0, copy_X=True, fit_intercept=True, max_iter=None,
   normalize=False, random_state=None, solver='auto', tol=0.001),
       fit_params=None, iid='warn', n_jobs=None,
       param_grid={'alpha': array([  1.     ,   1.49749, ...,  99.50251, 100.     ])},
       pre_dispatch='2*n_jobs', refit=True, return_train_score='warn',
       scoring='neg_mean_absolute_error', verbose=0)
```

```
grid_search.best_estimator_
```

```
Ridge(alpha=13.43718592964824, copy_X=True, fit_intercept=True, max_iter=None,
   normalize=False, random_state=None, solver='auto', tol=0.001)
```

this is the best estimator [with best value of λ], we can directly use this if we want or look at similar performance of other values and pick the one with least variance (most stable model) . Lets look at the custom function `Report` which will enable us to extract more detailed results from `grid_search.cv_results` object. Which is a huge dictionary containing information on performance of all combination of parameters that we are experiment with.

```
def report(results, n_top=3):
    for i in range(1, n_top + 1):
        # np.flatnonzero extracts index of `True` in a boolean array
        candidate = np.flatnonzero(results['rank_test_score'] == i)[0]
        # print rank of the model
        # values passed to function format here are put in the curly brackets when printing
```

```
        # 0 , 1 etc refer to placeholder for position of values passed to format function
        # .3f means upto 2 decimal digits
        print("Model with rank: {0}".format(i))
        # this prints cross validate performance and its standard deviation
        print("Mean validation score: {0:.5f} (std: {1:.5f})".format(
              results['mean_test_score'][candidate],
              results['std_test_score'][candidate]))
        # prints the paramter combination for which this performance was obtained
        print("Parameters: {0}".format(results['params'][candidate]))
        print("")
```

```
report(grid_search.cv_results_,3)
```

```
Model with rank: 1
Mean validation score: -1.60399 (std: 0.11152)
Parameters: {'alpha': 13.43718592964824}

Model with rank: 2
Mean validation score: -1.60399 (std: 0.11162)
Parameters: {'alpha': 13.93467336683417}

Model with rank: 3
Mean validation score: -1.60399 (std: 0.11171)
Parameters: {'alpha': 14.4321608040201}
```

Performance of these is pretty similar (looks identical due to us limiting this to 5 decimal digits) , there isn't much difference in stability either . We ca. go with the best guy. Tentative performance measure off by average 1.60 units .

if you want to make prediction , you can directly use `grid_search.predict` , it by defaults fits the model with best parameter choices on the entire data. However if you want to look at the coefficients, you'll have to fit the model separately.

```
ridge=grid_search.best_estimator_
```

```
ridge.fit(x_train,y_train)
```

```
Ridge(alpha=13.43718592964824, copy_X=True, fit_intercept=True, max_iter=None,
   normalize=False, random_state=None, solver='auto', tol=0.001)
```

```
ridge.intercept_
```

```
75.47046753682933
```

```
list(zip(x_train.columns,ridge.coef_))
```

```
[('Amount.Requested', 0.00016321376320470183),
 ('Debt.To.Income.Ratio', -0.0016649630043797535),
 ('Monthly.Income', -2.7907207451034204e-05),
 ('Open.CREDIT.Lines', -0.03648076179235473),
 ('Revolving.CREDIT.Balance', -2.683721318784198e-06),
 ('Inquiries.in.the.Last.6.Months', 0.3445032296978588),
 ('Employment.Length', 0.019601482704651712),
 ('fico', -0.08659781419319143),
 ('State_CA', -0.14115706864241717),
 ('State_NY', -0.10835699414412378),
 ('State_FL', -0.0112049655199493),
 ('State_TX', 0.4475943683966697),
 ('State_PA', -0.38750987879294396),
 ('State_IL', -0.47889963877418085),
 ('State_GA', -0.14818704990810666),
 ('State_NJ', -0.29109377437441625),
 ('State_VA', -0.05256231331588017),
 ('State_MA', -0.03973819224423696),
 ('State_NC', -0.342099870504766),
 ('State_OH', -0.202440726282369),
 ('State_MD', -0.023012728967286556),
 ('State_CO', 0.09019079073895374),
 ('Home.Ownership_MORTGAGE', -0.3571292206883735),
 ('Home.Ownership_RENT', -0.13033273805328766),
 ('Loan.Length_36 months', -3.013912446966353),
 ('Loan.Purpose_debt_consolidation', -0.41916847523850287),
 ('Loan.Purpose_credit_card', -0.5187875249317675),
 ('Loan.Purpose_other', 0.36593813217880045),
 ('Loan.Purpose_home_improvement', -0.3133150928985485),
 ('Loan.Purpose_major_purchase', -0.06049720536796199)]
```

you can see that there is no reduction in model coefficients . However if you compare them with coefficients obtained in simple linear regression without penalty, you'll find them that many of them have been suppressed by a good factor.

```
list(zip(x_train.columns,np.round(lr.coef_/ridge.coef_,2)))
```

```
[('Amount.Requested', 0.99),
 ('Debt.To.Income.Ratio', 3.13),
 ('Monthly.Income', 1.45),
 ('Open.CREDIT.Lines', 0.83),
 ('Revolving.CREDIT.Balance', 0.67),
 ('Inquiries.in.the.Last.6.Months', 0.95),
 ('Employment.Length', 1.19),
 ('fico', 1.01),
 ('State_CA', 1.15),
 ('State_NY', 1.33),
 ('State_FL', 10.46),
 ('State_TX', 1.0),
 ('State_PA', 2.41),
 ('State_IL', 0.85),
 ('State_GA', 2.24),
 ('State_NJ', 1.71),
 ('State_VA', 2.54),
 ('State_MA', 4.11),
 ('State_NC', 1.38),
 ('State_OH', 2.0),
 ('State_MD', 5.62),
 ('State_CO', 1.12),
 ('Home.Ownership_MORTGAGE', 1.58),
 ('Home.Ownership_RENT', 2.08),
 ('Loan.Length_36 months', 1.06),
 ('Loan.Purpose_debt_consolidation', 1.15),
 ('Loan.Purpose_credit_card', 1.1),
 ('Loan.Purpose_other', 0.96),
 ('Loan.Purpose_home_improvement', 1.58),
 ('Loan.Purpose_major_purchase', 3.95)]
```

Lets see if `Lasso` / `L1` penalty leads to model reduction or better performance .

we'll again do search for best λ for this

```
# there is no ideal range for any parameter search
# generally whatever range that you search in ,
# if the best value comes at the either edge of the range,
# you should expand on that side
# however if you keep on getting always largets value of lambda as best here
# that simply means that all vars are junk
# same on the lower side means , all vars are good and penalty is not necessary
lambdas=np.linspace(0.1,10,200)
params={'alpha':lambdas}
```

```
model=Lasso(fit_intercept=True)
```

```
grid_search=GridSearchCV(model,param_grid=params,cv=10,scoring='neg_mean_absolute_error')
```

```
grid_search.fit(x_train,y_train)
```

```
GridSearchCV(cv=10, error_score='raise-deprecating',
       estimator=Lasso(alpha=1.0, copy_X=True, fit_intercept=True, max_iter=1000,
   normalize=False, positive=False, precompute=False, random_state=None,
   selection='cyclic', tol=0.0001, warm_start=False),
       fit_params=None, iid='warn', n_jobs=None,
       param_grid={'alpha': array([ 0.1    ,  0.14975, ...,  9.95025, 10.  ])},
       pre_dispatch='2*n_jobs', refit=True, return_train_score='warn',
       scoring='neg_mean_absolute_error', verbose=0)
```

```
grid_search.best_estimator_
```

```
Lasso(alpha=0.1, copy_X=True, fit_intercept=True, max_iter=1000,
   normalize=False, positive=False, precompute=False, random_state=None,
   selection='cyclic', tol=0.0001, warm_start=False)
```

you can see that the best value came out to be on the lower end , we'll expand the range on that side .

```
lambdas=np.linspace(0.001,2,200)
params={'alpha':lambdas}
grid_search=GridSearchCV(model,param_grid=params,cv=10,scoring='neg_mean_absolute_error')
grid_search.fit(x_train,y_train)
```

```
GridSearchCV(cv=10, error_score='raise-deprecating',
       estimator=Lasso(alpha=1.0, copy_X=True, fit_intercept=True, max_iter=1000,
   normalize=False, positive=False, precompute=False, random_state=None,
   selection='cyclic', tol=0.0001, warm_start=False),
       fit_params=None, iid='warn', n_jobs=None,
       param_grid={'alpha': array([1.00000e-03, 1.10452e-02, ..., 1.98995e+00, 2.00000e+00])},
       pre_dispatch='2*n_jobs', refit=True, return_train_score='warn',
       scoring='neg_mean_absolute_error', verbose=0)
```

```
grid_search.best_estimator_
```

```
Lasso(alpha=0.011045226130653268, copy_X=True, fit_intercept=True,
   max_iter=1000, normalize=False, positive=False, precompute=False,
   random_state=None, selection='cyclic', tol=0.0001, warm_start=False)
```

this value is in between the range, we're good . Lets look at the cross-validate performance of some of the top few models

```
report(grid_search.cv_results_,3)
```

```
Model with rank: 1
Mean validation score: -1.60128 (std: 0.11647)
Parameters: {'alpha': 0.011045226130653268}

Model with rank: 2
Mean validation score: -1.60329 (std: 0.12013)
Parameters: {'alpha': 0.021090452261306535}

Model with rank: 3
Mean validation score: -1.60411 (std: 0.10911)
Parameters: {'alpha': 0.001}
```

There is no dramatic improment in performance , now lets see if their is any model reduction (β =0)

```
lasso=grid_search.best_estimator_
lasso.fit(x_train,y_train)
```

```
Lasso(alpha=0.011045226130653268, copy_X=True, fit_intercept=True,
   max_iter=1000, normalize=False, positive=False, precompute=False,
   random_state=None, selection='cyclic', tol=0.0001, warm_start=False)
```

```
list(zip(x_train.columns,lasso.coef_))
```

```
[('Amount.Requested', 0.00016023851703962787),
 ('Debt.To.Income.Ratio', -0.0009942600081876468),
 ('Monthly.Income', -2.7304512141858614e-05),
 ('Open.CREDIT.Lines', -0.036990231977433084),
 ('Revolving.CREDIT.Balance', -2.8844495569785946e-06),
 ('Inquiries.in.the.Last.6.Months', 0.33452489786480466),
 ('Employment.Length', 0.015521998102599638),
 ('fico', -0.08654353305995562),
 ('State_CA', -0.0),
 ('State_NY', -0.0),
 ('State_FL', 0.0),
 ('State_TX', 0.4213181413677211),
 ('State_PA', -0.07076519997728142),
 ('State_IL', -0.16988773003610938),
 ('State_GA', -0.0),
```

```
('State_NJ', -0.0),
('State_VA', 0.0),
('State_MA', 0.0),
('State_NC', -0.0),
('State_OH', -0.0),
('State_MD', 0.0),
('State_CO', 0.0),
('Home.Ownership_MORTGAGE', -0.2085859940218407),
('Home.Ownership_RENT', -0.0),
('Loan.Length_36 months', -3.074433736924612),
('Loan.Purpose_debt_consolidation', -0.2540228534857545),
('Loan.Purpose_credit_card', -0.32972568683630343),
('Loan.Purpose_other', 0.3971422404793555),
('Loan.Purpose_home_improvement', -0.022332982889820444),
('Loan.Purpose_major_purchase', 0.0)]
```

you can see that many of the coefficients have become exactly zero . Much smaller model gives you similar performance . In this case, amongst all, Lasso model is the best , considering its size and performance. That doesn't mean, Lasso or L1 penalty will always result in best model. It depends on the data.

`Note : Ridge and Lasso regression are just different ways of estimating the parameters. Eventual prediction model is linear in both as well as in simple linear regression`

Lets build a classification linear model

You have seen how to model a continuous numeric response with linear regression technique. But in many business scenarios our target is binary. For example whether someone will buy my product , whether someone will default on the loan they have taken. Answer to all these and many other such questions is yes/no. Here the output variable values are discrete & finite rather than continuous & infinite values like in Linear Regression. For analysing such type of data, we could try to frame a rule which helps in guessing the outcome from the input variables. This is called a classification problem which is an important topic in statistics and machine learning. classification is the problem of identifying to which of a set of categories a new observation belongs, on the basis of a training set of data containing observations whose category membership is known. Some examples of Classification Tasks are listed below:

- In medical field, the classification task could be assigning a diagnosis to a given patient as described by observed characteristics of the patient such as age, gender, blood pressure, body mass index, presence or absence of certain symptoms, etc.
- In banking sector, one may want to categorize hundreds or thousands of applications for new cards containing information for several attributes such as annual salary, outstanding debts, age etc., into users who have good credit or bad credit for enabling a credit card company to do further analysis for decision making; OR one might want to learn to predict whether a particular credit card charge is legitimate or fraudulent.
- In social sciences, we may be interested to predict the preference of a voter for a party

based on : age, income, sex, race, residence state, votes in previous elections etc.
- In finance sector, one would require to ascertain , whether a vendor is credit worthy?
- In insurance domain, the company will need to assess , Is the submitted claim fraudulent or genuine?
- In Marketing, the marketer would like to figure out , Which segment of consumers are likely to buy?

All of the problems listed above use same underlying algorithms, despite them being pretty different from each other on the face of it.

In this module we'll look at where , $f(X)$ is linear combination of variables as discussed earlier . Here is the problem statement we'll be working on

Problem Statement :

A financial institution is planning to roll out a stock market trading facilitation service for their existing account holders. This service costs significant amount of money for the bank in terms of infra, licensing and people cost. To make the service offering profitable, they charge a percentage base commission on every trade transaction. However this is not a unique service offered by them, many of their other competitors are offering the same service and at lesser commission some times. To retain or attract people who trade heavily on stock market and in turn generate a good commission for institution, they are planning to offer discounts as they roll out the service to entire customer base.

Problem is , that this discount, hampers profits coming from the customers who do not trade in large quantities . To tackle this issue , company wants to offer discounts selectively. To be able to do so, they need to know which of their customers are going to be heavy traders or money makers for them.

To be able to do this, they decided to do a beta run of their service to a small chunk of their customer base [approx 10000 people]. For these customers they have manually divided them into two revenue categories 1 and 2. Revenue one category is the one which are money makers for the bank, revenue category 2 are the ones which need to be kept out of discount offers.

We need to use this study's data to build a prediction model which should be able to identify if a customer is potentially eligible for discounts [falls In revenue grid category 1]. Lets get the data and begin.

Classification Model Evaluation and Probability Scores to Hard Classes

Before we jump-in head first into building our model, we need to figure out couple of things about classification problems in general . We discussed earlier that prediction model for classification output probabilities as outcome. In many case that'll suffice as a way of scoring our observation in terms most likely to least likely. However in many cases we'd eventually need to convert those probability scores to hard classes.

The way is to figure out a cut-off/threshold for the probability scores where we can say that, obs with higher score than this will be classified as 1 and others will be classified as 0. Now, the question becomes; how do we come up with this cut-off.

Confusion Matrix and associated measurements

Irrespective of what cut-off we chose, the hard class prediction rule is fixed. if probability score is higher than the cutoff then the prediction is 1 otherwise 0 [given, we are predicting probability of outcome being 1]. Any cut-off decision results in some of our predictions being true and some of them being false. Since there are are only two hard classes, we'll have 4 possible cases .

- When we predict 1 [+ve] but in reality the outcome is 0[-ve] : False Positive
- When we predict 0 [-ve] but in reality the outcome is 1[+ve] : False Negative
- When we predict 1 [+ve] and in reality the outcome is 1[+ve] : True Positive
- When we predict 0 [-ve] and in reality the outcome is 0[-ve] : True Positive

This is generally displayed in a matrix format known as confusion matrix .

**	Positive_predicted	Negative_predicted
Positive_real	**TP**	**FN**
Negative_real	**FP**	**TN**

TP : Number of true positive cases
TN : Number of true negative cases
FP : Number of false positive cases
FN : Number of false negative cases

Using these figures we can come up with couple of popular measurements in context of classification

How accurate our predictions are

Accuracy = $\frac{TP+TN}{\text{Total obs}}$

How well we are capturing/recalling positive cases

Sensitivity or Recall = $\frac{TP}{TP+FN}$

How well we are capturing negative cases

Specificity = $\frac{TN}{TN+FP}$

How accurately we have been able to predict positive cases

Precision = $\frac{TP}{TP+FP}$

Some might suggest that we can take any one of them, and measure for all scores , and decides that score to be our cutoff where the taken measurement is highest for the training data. However none of these above mentioned measurements take care of our business requirement of good separation between two classes. Here are the issues associated with these individually if we consider them as candidate for determining cutoffs .

Accuracy This works only if our classes are roughly equally present in the data. However generally this is not the case in many business problems. For example, consider campaign response model. Typical response rate is 0.5-2%. Even if predict that none of the customers are going to subscribe to our campaign; accuracy will be in the range 98-99.5%, quite misleading.

Sensitivity or Recall We can arbitrarily make Sensitivity/Recall 100% by predicting all the cases as positive . Which is kinda useless because it doesn't take into account that labeling lot of negative cases as positive should be penalized too.

Specificity We can arbitrarily make Specificity 100% by predicting all the cases as Negative . Which is kinda useless because it doesn't take into account that labeling lot of positive cases as negative should be penalized too.

Precision We can make precision very high by keep cut-off too high and thus predicting very few sure shot cases as positive, but in that scenario our recall will be down to dumps.

So it turns out that, these measures are a good look into how our model is doing , but none of them taken individually can be used to determine proper cut-off. Following are some proper measures which give equal weight to goal of capturing as many positives as possible and at the same time; not labeling too many negative cases as positives .

KS = $\frac{TP}{TP+FN} - \frac{FP}{TN+FP}$

You can see that `KS` will be highest when we recall is high but at the same time we are not labeling a lot of negative cases as positive. We can calculate `KS` for all the scores and chose that score as our ideal cut-off for which `KS` is maximum.

There is another measure which lets you give different weights for your precision or recall. There can be cases in real business problems where you'd want to give more importance to recall over precision or otherwise. Consider a critical test which determines whether someone has a particular aggressively fatal decease or not . In this case we wouldn't want to miss out on positive cases and wouldn't really mind some of the negative cases being labeled as positive.

F_β Score = $\frac{(1+\beta^2)*Precision*Recall}{(\beta^2*Precision)+Recall}$

Notice that value of β here will determine; how much and what are we going to give importance to. For β=1 , equal importance is given to both precision and recall. When 1> $\beta \to 0$, F_β score favors Precision and results in high probability score being chosen as cutoff. On the other hand when β > 1 and as $\beta \to \infty$ it favors Recall and results in low probability scores being choses as cutoff.

Now these let us find a proper cutoff given a probability score model. However so far we haven't discussed how do asses , how good the score is itself . Next section is dedicated to the same .

ROC curve and AUC Score

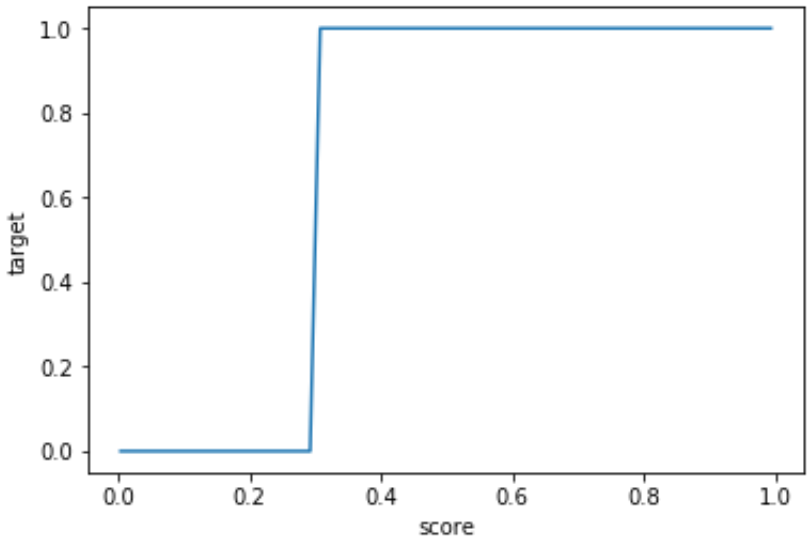
When we asses performance of a classification probability score , we try to see how it stacks up with an ideal scenario. For an ideal score, there will exist a clean cutoff; meaning, there will be no overlap between two classes when chose that cut-off. There will be no `False Positives` or `False Negatives`

Consider a hypothetical scenario where we have an ideal prob score like given below .

```
d=pd.DataFrame({'score':np.random.random(size=100)})
d['target']=(d['score']>0.3).astype(int)
```

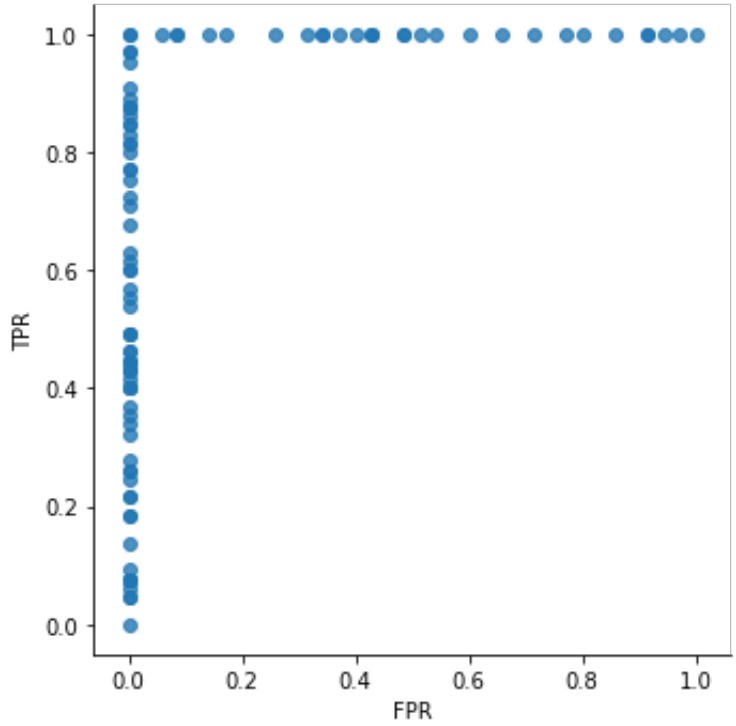
```
import seaborn as sns
```

```
sns.lineplot(x='score',y='target',data=d)
```



For this ideal scenario, we are going to consider many cutoffs between 0-1 and calculate `True Positive Rate` [Same as Sensitivity] and `False Positive Rate` [Same as 1-Specificity]. And plot those . The resultant plot that we'll get is known as ROC Curve . Lets see how that looks

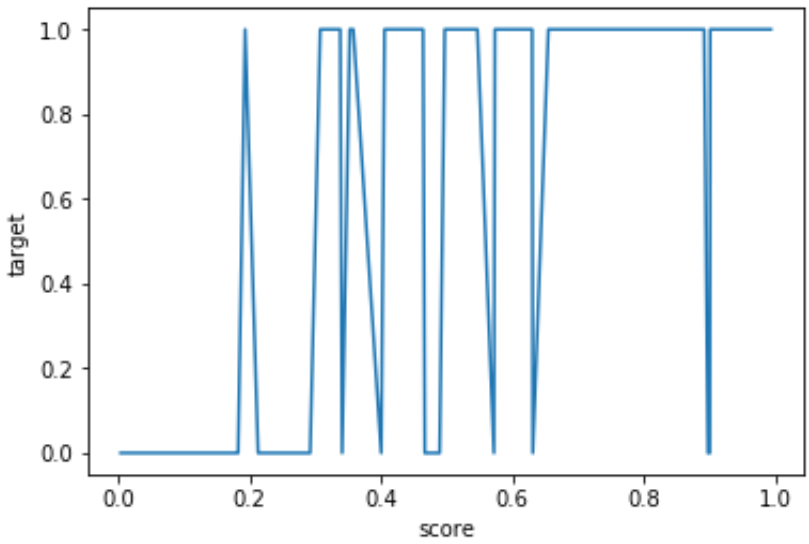
```
TPR=[]
FPR=[]
real=d['target']
for cutoff in np.linspace(0,1,100):
    predicted=(d['score']>cutoff).astype(int)
    TP=((real==1)&(predicted==1)).sum()
    FP=((real==0)&(predicted==1)).sum()
    TN=((real==0)&(predicted==0)).sum()
    FN=((real==1)&(predicted==0)).sum()

    TPR.append(TP/(TP+FN))
    FPR.append(FP/(TN+FP))

temp=pd.DataFrame({'TPR':TPR,'FPR':FPR})

sns.lmplot(y='TPR',x='FPR',data=temp,fit_reg=False)
```



Its perfectly triangular , and area under the curve is 1 for this ideal scenario, however lets see what happens if we make it not so ideal scenario and add some overlap. [we'll flip some targets in the same data]

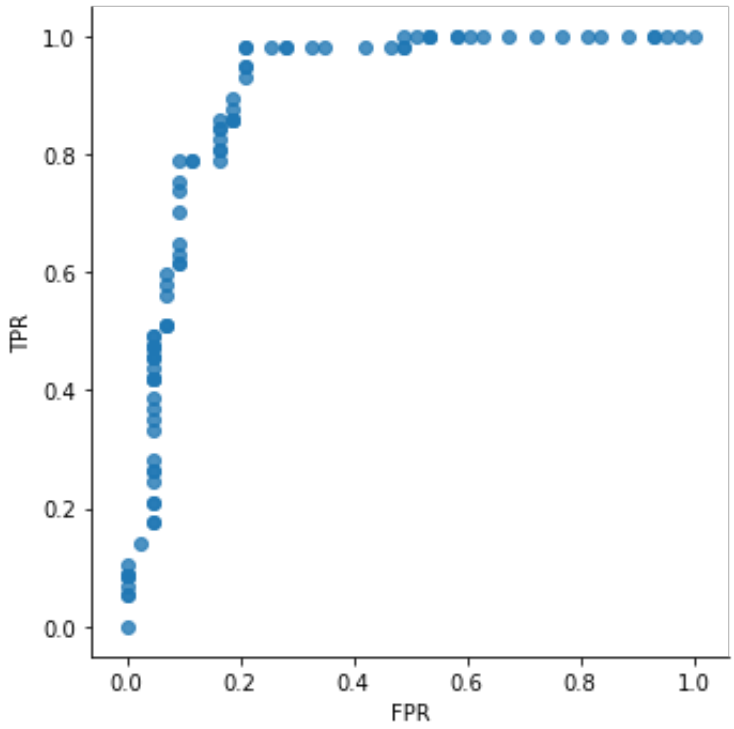
```
inds=np.random.choice(range(100),10,replace=False)
d.iloc[inds,1]=1-d.iloc[inds,1]
```

```
sns.lineplot(x='score',y='target',data=d)
```



you can see that there is lot of overlap now and , there can not exist a clear cutoff. Lets see how the ROC curve looks for the same

```
TPR=[]
FPR=[]
real=d['target']
for cutoff in np.linspace(0,1,100):
    predicted=(d['score']>cutoff).astype(int)
    TP=((real==1)&(predicted==1)).sum()
    FP=((real==0)&(predicted==1)).sum()
    TN=((real==0)&(predicted==0)).sum()
    FN=((real==1)&(predicted==0)).sum()

    TPR.append(TP/(TP+FN))
    FPR.append(FP/(TN+FP))

temp=pd.DataFrame({'TPR':TPR,'FPR':FPR})

sns.lmplot(y='TPR',x='FPR',data=temp,fit_reg=False)
```



You can try introducing more overlap and you'll see the ROC curve move away from the ideal scenario. Area Under the Curve(AUC Score) will become less than one, more close to 1 it is more close to ideal scenario it is, better is your model . Now that we have a way to asses our model , lets begin building it

```
# Here we we'll demonstrate our earlier suggestion, where we combine train
# test data and do data prep so that both of them have same set of
variables
```

```
train_file=r'~/Dropbox/0.0 Data/rg_train.csv'
test_file=r'~/Dropbox/0.0 Data/rg_test.csv'
bd_train=pd.read_csv(train_file)

bd_test=pd.read_csv(test_file)
bd_train['data']='train'
bd_test['data']='test'
bd_all=pd.concat([bd_train,bd_test],axis=0)
```

These are few data decision that we have taken after exploring the data Feel free to go alternate routes and see how that makes a difference to model performance

```
# REF_NO,post_code , post_area  : drop [ too many unique values]
# children : Zero : 0 , 4+ : 4 and then convert to numeric
# age_band, family income : string processing and then to numeric
# status , occupation , occupation_partner , home_status,
# self_employed ,TVArea , Region , gender : dummies
# Revenue Grid : 1,2 : 1,0 [some functions need the target to be 1/0 in binary classfication]
```

```
bd_all.drop(['REF_NO','post_code','post_area'],axis=1,inplace=True)
```

```
bd_all['children']=np.where(bd_all['children']=='Zero',0,bd_all['children'])
bd_all['children']=np.where(bd_all['children'].str[:1]=='4',4,bd_all['children'])
bd_all['children']=pd.to_numeric(bd_all['children'],errors='coerce')
```

```
bd_all['Revenue.Grid']=(bd_all['Revenue.Grid']==1).astype(int)
```

```
bd_all['family_income'].value_counts(dropna=False)
```

```
>=35,000             2517
<27,500, >=25,000    1227
<30,000, >=27,500     994
<25,000, >=22,500     833
<20,000, >=17,500     683
<12,500, >=10,000     677
<17,500, >=15,000     634
<15,000, >=12,500     629
<22,500, >=20,000     590
<10,000, >= 8,000     563
< 8,000, >= 4,000     402
```

```
< 4,000               278
Unknown               128
Name: family_income, dtype: int64
```

```
bd_all['family_income']=bd_all['family_income'].str.replace(',',"")
bd_all['family_income']=bd_all['family_income'].str.replace('<',"")
k=bd_all['family_income'].str.split('>=',expand=True)
```

```
for col in k.columns:
    k[col]=pd.to_numeric(k[col],errors='coerce')
```

```
bd_all['fi']=np.where(bd_all['family_income']=='Unknown',np.nan,
                      np.where(k[0].isnull(),k[1],
                      np.where(k[1].isnull(),k[0],0.5*(k[0]+k[1]))))
```

```
bd_all['age_band'].value_counts(dropna=False)
```

```
45-50      1359
36-40      1134
41-45      1112
31-35      1061
51-55      1052
55-60      1047
26-30       927
61-65       881
65-70       598
22-25       456
71+         410
18-21        63
Unknown      55
Name: age_band, dtype: int64
```

```
k=bd_all['age_band'].str.split('-',expand=True)
for col in k.columns:
    k[col]=pd.to_numeric(k[col],errors='coerce')
```

```
bd_all['ab']=np.where(bd_all['age_band'].str[:2]=='71',71,
                      np.where(bd_all['age_band']=='Unknow',np.nan,0.5*
(k[0]+k[1])))
```

```
del bd_all['age_band']
del bd_all['family_income']
```

```
cat_vars=bd_all.select_dtypes(['object']).columns
cat_vars=list(cat_vars)
cat_vars.remove('data')
# we are using pd.get_dummies here to create dummies
# its more straight forward but doesnt let you ignore categories on the basis of frequencies
for col in cat_vars:
    dummy=pd.get_dummies(bd_all[col],drop_first=True,prefix=col)
    bd_all=pd.concat([bd_all,dummy],axis=1)
    del bd_all[col]
    print(col)
del dummy
```

```
TVarea
gender
home_status
occupation
occupation_partner
region
self_employed
self_employed_partner
status
```

```
# imputing missing values
for col in bd_all.columns:
    if col=='data' or bd_all[col].isnull().sum()==0:continue

 bd_all.loc[bd_all[col].isnull(),col]=bd_all.loc[bd_all['data']=='train',col].mean()

```

```
# separating data
bd_train=bd_all[bd_all['data']=='train']
del bd_train['data']
bd_test=bd_all[bd_all['data']=='test']
bd_test.drop(['Revenue.Grid','data'],axis=1,inplace=True)

```

In scikit-learn L1 and L2 penalties are implemented within the funcition `LogisticRegression` it self. We'll be treating them as parameters to tune with cross validation. The counterpart to λ is C, which is implemented in a way that , lower the value of C, higher the penalty . There is another parameter `class_weight`, takes two values `balanced` and `None`. `None` here implies equal weight to all observation while calculating the cost/loss [thus all observation having equal contribution to loss, this might make the model biased towards the majority class if their is class imbalance] . `balanced` artificially inflates weight of the minority class [class with low frequency] in order to ensure that cost/loss contribution is same from both the classes and model is focused on separation of the classes .

```
from sklearn.linear_model import LogisticRegression
```

```
params={'class_weight':['balanced',None],
        'penalty':['l1','l2'],
        # these are L1 and L2 written in lower case
        # dont confuse them with numeric eleven and tweleve
        'C':np.linspace(0.0001,1000,10)}
# we can certainly try much higher ranges and number of values for the parameter 'C'
# grid search in this case , will be trying out 2*2*10=40 possible combination
# and will give us cross validated performance for all
```

```
model=LogisticRegression(fit_intercept=True)
```

```
from sklearn.model_selection import GridSearchCV
```

```
grid_search=GridSearchCV(model,param_grid=params,cv=10,scoring="roc_auc",n_jobs=-1)
# note that scoring is now roc_auc as we are solving a classification problem
# n_jobs has nothing to do with model building as such
# it enables parallel processing , number reflects number of cores
# of your processor being utilised. -1 , means all the cores
```

```
x_train=bd_train.drop('Revenue.Grid',axis=1)
y_train=bd_train['Revenue.Grid']
```

```
grid_search.fit(x_train,y_train)
```

```
GridSearchCV(cv=10, error_score='raise-deprecating',
       estimator=LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
          intercept_scaling=1, max_iter=100, multi_class='warn',
          n_jobs=None, penalty='l2', random_state=None, solver='warn',
          tol=0.0001, verbose=0, warm_start=False),
       fit_params=None, iid='warn', n_jobs=-1,
       param_grid={'class_weight': ['balanced', None], 'penalty': ['l1', 'l2'], 'C': array([1.00000e-04, 1.11111e+02, 2.22222e+02, 3.33333e+02, 4.44445e+02,
       5.55556e+02, 6.66667e+02, 7.77778e+02, 8.88889e+02, 1.00000e+03])},
       pre_dispatch='2*n_jobs', refit=True, return_train_score='warn',
       scoring='roc_auc', verbose=0)
```

```
report(grid_search.cv_results_,5)
```

```
Model with rank: 1
Mean validation score: 0.95466 (std: 0.01253)
Parameters: {'C': 0.0001, 'class_weight': 'balanced', 'penalty': 'l2'}

Model with rank: 2
Mean validation score: 0.95269 (std: 0.01273)
Parameters: {'C': 555.5556, 'class_weight': 'balanced', 'penalty': 'l2'}

Model with rank: 3
Mean validation score: 0.95255 (std: 0.01325)
Parameters: {'C': 444.4445, 'class_weight': 'balanced', 'penalty': 'l2'}

Model with rank: 4
Mean validation score: 0.95236 (std: 0.01232)
Parameters: {'C': 888.8889, 'class_weight': 'balanced', 'penalty': 'l2'}

Model with rank: 5
Mean validation score: 0.95214 (std: 0.01210)
Parameters: {'C': 777.7778000000001, 'class_weight': 'balanced', 'penalty': 'l2'}
```

We'll go ahead with the best model here . Although ideally we should expand the range on C on the lower side and run the experiment as the best values is coming at the edge. I am leaving that for you to try . Since this is with `l2` penalty there will not be any model reduction

if we want to make prediction just for probabilities and submit , we can simply use `grid_search` object. As for the tentative performance of this model, we can already see that in the outcome of report function with cross validated auc acore

```
# predict_proba for predciting probabilities
# just predict, predicts hard classes considering 0.5 as score cutoff
# which is not always a great idea, we'll see in a moment
# how to determine our own cutoff, in case we need to predict hard classes
test_prediction = grid_search.predict_proba(bd_test)
```

```
test_prediction
```

```
array([[0.99496285, 0.00503715],
       [0.95418665, 0.04581335],
       [0.98695233, 0.01304767],
       ...,
       [0.97058409, 0.02941591],
       [0.77185663, 0.22814337],
       [0.80655962, 0.19344038]])
```

Note that this gives two probabilities for each observation and they sum up to 1

```
# this will tell you which probability belongs to which class
grid_search.classes_
```

```
array([0, 1])
```

this means first probability is for the outcome being 0 and second is for the outcome being 1

you can extract probability for either class by using proper index

- `test_prediction[:,0]` : for probability of outcome being 0
- `test_prediction[:,1]` : for probability of outcome being 1

you can submit this by , converting it to a pandas data frame and then using function `to_csv` to write it to a csv file

Finding cutoff on the basis of max KS

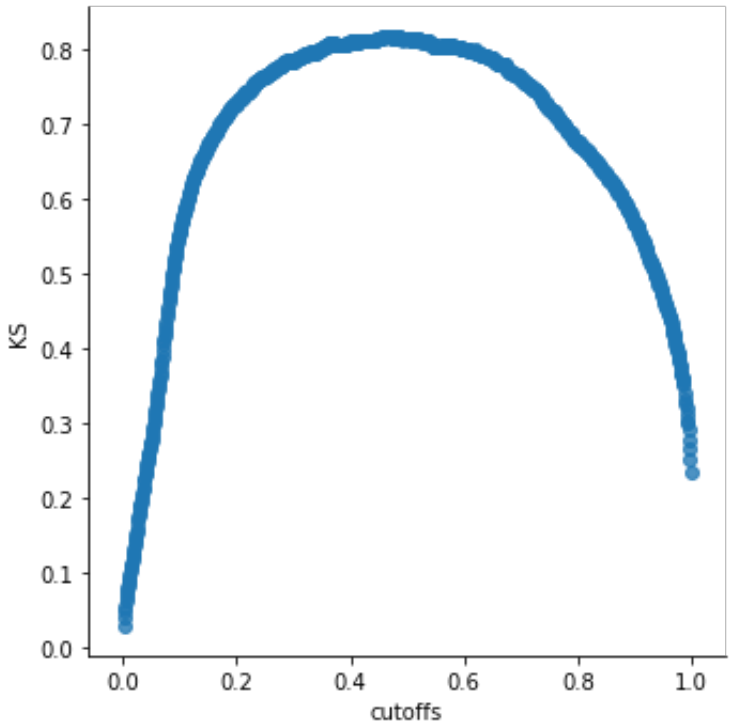
```
train_score=grid_search.predict_proba(x_train)[:,1]
real = y_train
```

```
cutoffs = np.linspace(.001,0.999, 999)
```

```
KS=[]
```

```
for cutoff in cutoffs:
    predicted=(train_score>cutoff).astype(int)
    TP=((real==1)&(predicted==1)).sum()
    FP=((real==0)&(predicted==1)).sum()
    TN=((real==0)&(predicted==0)).sum()
    FN=((real==1)&(predicted==0)).sum()

    ks=(TP/(TP+FN))-(FP/(TN+FP))
    KS.append(ks)
```

```
temp=pd.DataFrame({'cutoffs':cutoffs,'KS':KS})
sns.lmplot(x='cutoffs',y='KS',data=temp,fit_reg=False)
```



We can now find for which cutoff value KS takes its maximum value

```
cutoffs[KS==max(KS)][0]
```

```
0.467
```

if we have to submit hardclass we'll use this cutoff to convert probability score to hard classes

```
test_hard_classes=(test_prediction>cutoffs[KS==max(KS)][0]).astype(int)
```

and write this to csv to submit

Before we conclude this module, few key takeaways :

- Data Prep part will remains same , irrespective of what modules we study further
- Performance measures and methods will also remain same irrespective of the algorithm
- The way to find cutoffs for probability score will also remain same irrespective what which algorithm we obtain those scores from