**TITANIC PROJECT REPORT:**

**Problem Definition:**

In this article, we will go through the Titanic problem is based on the sinking of the ‘Unsinkable’ ship Titanic in early 1912. during her maiden voyage, the Titanic sank after colliding with an iceberg, killing 1502 out of 2224 passengers and crew. This sensational tragedy shocked the international community and led to better safety regulations for ships.

One of the reasons that the shipwreck led to such loss of life was that there were not enough lifeboats for the passengers and crew. Although there was some element of luck involved in surviving the sinking, some groups of people were more likely to survive than others, such as women, children, and the upper-class.

**Data Analysis:**

We have to make sure we update the categorical variables to numerical variables as the machine learning techniques we will apply requires all passengers survival data's to be numerical. We can see our dataset but we also want to make sure the data is clean, so as part of the cleaning process, we look at missing values and data types.

The dataset has a lot of categorical variables rendered as text values (‘Yes’,’No’,etc.) in the columns. Converting the categorical values into numeric values to facilitate ML algorithms to process the data. Also, converting Total Charges to a numerical data type.

df = df.drop(['Ticket','Cabin'], axis=1)

*# Remove NaN values*

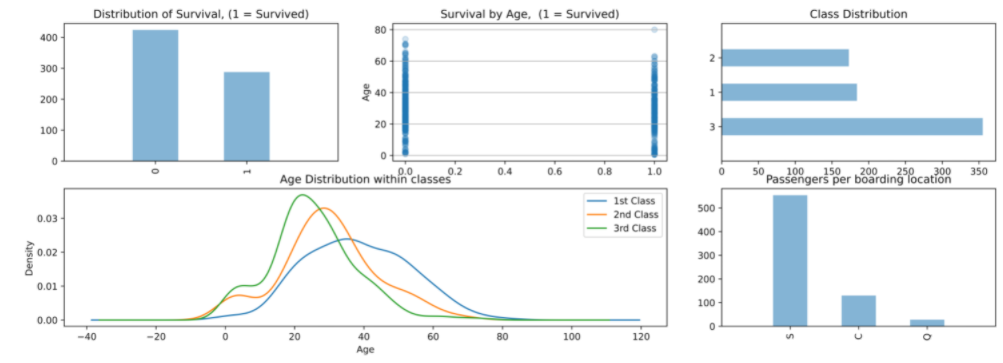
df = df.dropna()

**Explanatory data analysis:**

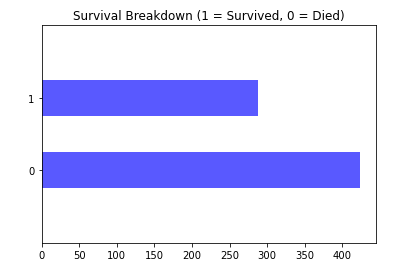
The explanatory approach is a method to make people understand something by describing or illustrating. In this stage we able to understand and finds the overview of the data headings, data shapes, data types, unique values, missing values, perform basic statistics, univariate and bivariate anaysis , missing value treatment and outliers treatment.

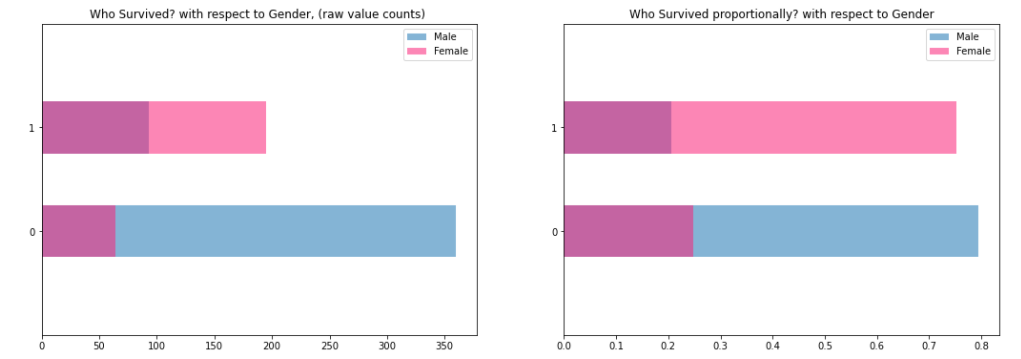
Majorly here to predict if an individual will survive based on the features in the data like:

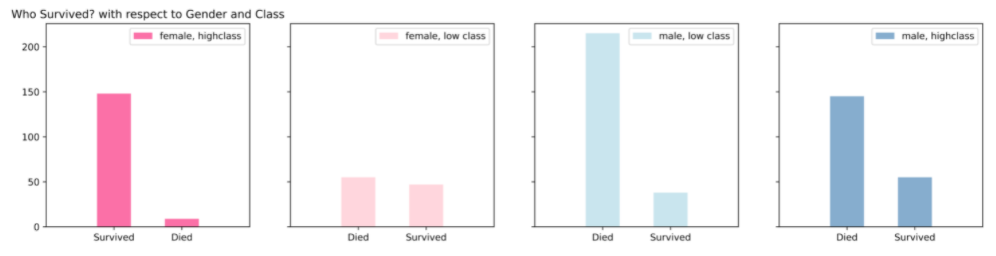
* Traveling Class (called pclass in the data)
* Sex
* Age
* Fare Price

 Traveling Class (called pclass in the data) Sex Age Fare Price Let’s see if we can gain a better understanding of who survived and died.

First let’s plot a bar graph of those who Survived Vs. Those who did not.







**Data preprocessing:**

Data as such can't be used. We'll transform the data so that we can feed it to a machine learning algorithm

**import statsmodels.api as sm**

**from statsmodels.nonparametric.kde import KDEUnivariate**

**from statsmodels.nonparametric import smoothers\_lowess**

**from pandas import Series, DataFrame**

**from patsy import dmatrices**

**from sklearn import datasets, svm**

### Methodology:

Our first step was to split our data into training and test sets using train-test-split, which would allow us to cross-validate our results later. We also stratified the train-test-split, to ensure that the same proportion of our target variable was found in both our training and test sets. It is important to test a predictor on data held-out from training, preprocessing (such as standardization, feature selection, etc.)

### Splitting the data:

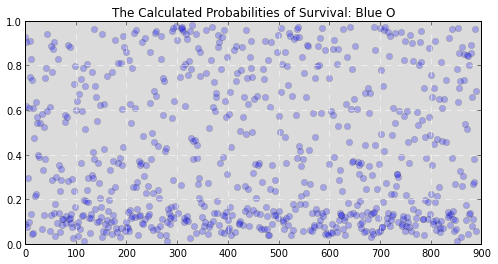
X is the data with the independent variables, Y is the data with the dependent variable. The test size variable determines in which ratio the data will be split. It is quite common to do this in a 80 Training / 20 Test ratio. Also ned to stratify the train-test-split to have a balanced split.

**Building Machine Learning Models:**

## Supervised Machine Learning:

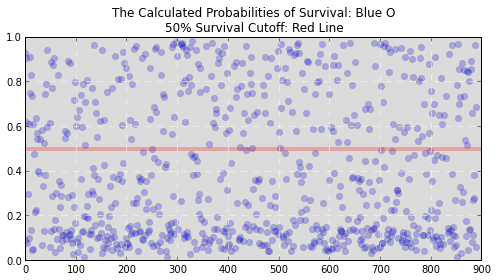
#### Logistic Regression:

Our aim is to predict a binary outcome. That is, it wants to know whether some will die, (represented as a 0), or survive, (represented as 1). A good place to start is to calculate the probability that an individual observation, or person, is likely to be a 0 or 1. That way we would know the chance that someone survives, and could start making somewhat informed predictions. If we did, we'd get results like this::



(Y axis is the probability that someone survives, X axis is the passenger’s number from 1 to 891.)

While that information is useful it doesn’t let us know whether someone ended up alive or dead. It just lets us know the chance that they will survive or die. We still need to translate these probabilities into the binary decision we’re looking for. But how? We could arbitrarily say that our survival cutoff is anyone with a probability of survival over 50%. In fact, this tactic would actually perform pretty well for our data and would allow you to make decently accurate predictions. Graphically it would look something like this:



If you’re a betting man like me, you don’t like to leave everything to chance. What are the odds that setting that cutoff at 50% works? Maybe 20% or 80% would work better. Clearly we need a more exact way to make that cutoff. What can save the day? In steps the **Logistic Regression**.

A logistic regression follows the all steps we took above but mathematically calculates the cutoff, or decision boundary (as stats nerds call it), for you. This way it can figure out the best cut off to choose, perhaps 50% or 51.84%, that most accurately represents the training data.

The three cells below show the process of creating our Logitist regression model, training it on the data, and examining its performance.

First, we define our formula for our Logit regression. In the next cell we create a regression friendly dataframe that sets up boolean values for the categorical variables in our formula and lets our regression model know the types of inputs we're giving it. The model is then instantiated and fitted before a summary of the model's performance is printed. In the last cell we graphically compare the predictions of our model to the actual values we are trying to predict, as well as the residual errors from our model to check for any structure we may have missed.

*# create a regression friendly dataframe using patsy's dmatrices function*

y,x = dmatrices(formula, data=df, return\_type='dataframe')

*# instantiate our model*

model = sm.Logit(y,x)

*# fit our model to the training data*

res = model.fit()

*# save the result for outputing predictions later*

results['Logit'] = [res, formula]

res.summary()

### Support Vector Machine (SVM):

The logit model we just implemented was great in that it showed exactly where to draw our decision boundary or our 'survival cut off'. A linear line is okay, but can we do better? Perhaps a more complex decision boundary like a wave, circle, or maybe some sort of strange polygon would describe the variance observed in our sample better than a line. Imagine if we were predicating survival based on age. It could be a linear decision boundary, meaning each additional time you've gone around the sun you were 1 unit more or less likely to survive. But I think it could be easy to imagine some sort of curve, where a young healthy person would have the best chance of survival, and sadly the very old and very young a like: a poor chance. Now that’s a interesting question to answer. But our logit model can only evaluate a linear decision boundary. How do we get around this? With the usual answer to life the universe and everything; .

**The answer:** We could transform our logit equation from expressing a linear relationship like so:

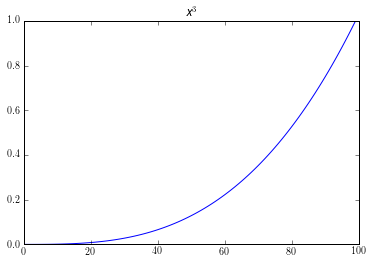
$survived = \beta\_0 + \beta\_1pclass + \beta\_2sex + \beta\_3age + \beta\_4sibsp + \beta\_5parch + \beta\_6embarked$

Which we'll represent for convenience as:

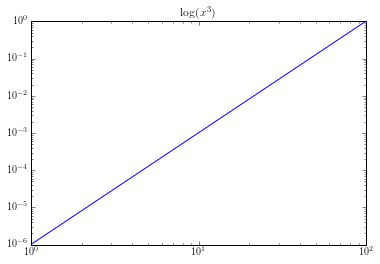
to a expressing a linear expression of a non-linear relationship:

By doing this we're not breaking the rules. Logit models are only efficient at modeling linear relationships, so we're just giving it a linear relationship of a non-linear thing.

An easy way to visualize this by looking at a graph an exponential relationship. Like the graph of :



Here its obvious that this is not linear. If used it as an equation for our logit model, ; we would get bad results. But if we transformed it by taking the log of our equation, . We would get a graph like this:



That looks pretty linear to me.

This process of transforming models so that they can be better expressed in a different mathematical plane is exactly what the Support Vector Machine does for us.

**Cross-validation:**

Cross validation ensures all samples will appear in the training and test sets, so 100% of your data gets used at some point for training and for testing. Use 5-fold cross-validation instead, and take the average of all of my data to make the calibration plot.

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

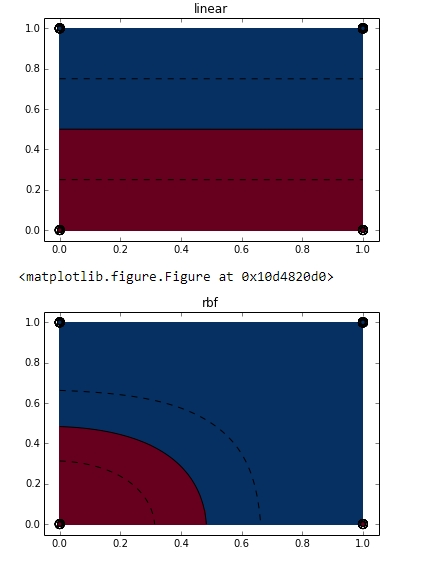
nighty\_precent\_of\_sample = int(.9 \* n\_sample)

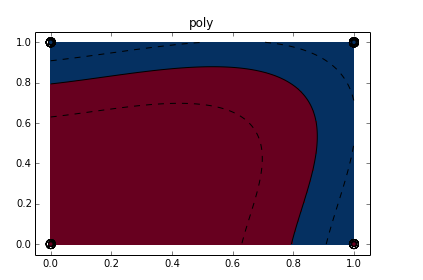
X\_train = X[:nighty\_precent\_of\_sample]

y\_train = y[:nighty\_precent\_of\_sample]

X\_test = X[nighty\_precent\_of\_sample:]

y\_test = y[nighty\_precent\_of\_sample:]





Any value in the blue survived while anyone in the read did not. Checkout the graph for the linear transformation. It created its decision boundary right on 50%! That guess from earlier turned out to be pretty good. As you can see, the remaining decision boundaries are much more complex than our original linear decision boundary. These more complex boundaries may be able to capture more structure in the dataset, if that structure exists, and so might create a more powerful predictive model.

### Random Forest:

Random forests are an ensemble learning method for classification (and regression) that operate by constructing a multitude of decision trees at training time and outputting the class that is the mode of the classes output by individual trees.

There are always skeptics, and you just might be one about all the fancy lines we've created so far. Well for you, here’s another option; the Random Forest. This technique is a form of non-parametric modeling that does away with all those equations we created above, and uses raw computing power and a clever statistical observation to tease the structure out of the data.

**How do they work?** A random forest algorithm randomly generates many extremely simple models to explain the variance observed in random subsections of our data. These models are like our gumball guesses. They are all awful individually. Really awful. But once they are averaged, they can be powerful predictive tools. The averaging step is the secret sauce. While the vast majority of those models were extremely poor; they were all as bad as each other on average. So when their predictions are averaged together, the bad ones average their effect on our model out to zero. The thing that remains, if anything, is one or a handful of those models have stumbled upon the true structure of the data. The cell below shows the process of instantiating and fitting a random forest, generating predictions form the resulting model, and then scoring the results.

*# import the machine learning library that holds the randomforest*

**import** **sklearn.ensemble** **as** **ske**

*# Create the random forest model and fit the model to our training data*

y, x = dmatrices(formula\_ml, data=df, return\_type='dataframe')

*# RandomForestClassifier expects a 1 demensional NumPy array, so we convert*

y = np.asarray(y).ravel()

*#instantiate and fit our model*

results\_rf = ske.RandomForestClassifier(n\_estimators=100).fit(x, y)

*# Score the results*

score = results\_rf.score(x, y)

print "Mean accuracy of Random Forest Predictions on the data was: **{0}**".format(score)

**Concluding Remarks:**

Mean accuracy of Random Forest Predictions on the data was: 0.945224719101

Our random forest performed only slightly better than a thumb wave, meaning that if you randomly assigned 1s and 0s by waving your thumb up and down you would do almost as well on average. It seems that this time our random forest did not stumble on the true structure of the data.