                     **TITANIC DISASTER AND MACHINE LEARNING**

Here I am going to make an analysis on one of the worst ship disasters in human history and build a model. This model will give various informations about the passengers like(Age,Sex,Class and so on). This will also make a prediction about the possibility/probability of a passenger to have survived the disaster.

**Backdrop**

The RMS Titanic was a British passenger liner that sank in the North Atlantic Ocean in the early morning hours of 15 April 1912, after it collided with an iceberg during its maiden voyage from Southampton to New York City. There were an estimated 2,224 passengers and crew aboard the ship, and more than 1,500 died, making it one of the deadliest commercial peacetime maritime disasters in modern history. The RMS Titanic was the largest ship afloat at the time it entered service and was the second of three Olympic-class ocean liners operated by the White Star Line. The Titanic was built by the Harland and Wolff shipyard in Belfast. Thomas Andrews, her architect, died in the disaster.

# **Importing the Libraries**

*# linear algebra*

**import** **numpy** **as** **np**

*# data processing*

**import** **pandas** **as** **pd**

*# data visualization*

**import** **seaborn** **as** **sns**

%matplotlib inline

**from** **matplotlib** **import** pyplot **as** plt

**from** **matplotlib** **import** style

*# Algorithms*

**from** **sklearn** **import** linear\_model

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

**from** **sklearn.ensemble** **import** RandomForestClassifier

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

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

**from** **sklearn.tree** **import** DecisionTreeClassifier

**from** **sklearn.neighbors** **import** KNeighborsClassifier

**from** **sklearn.svm** **import** SVC, LinearSVC

**from** **sklearn.naive\_bayes** **import** GaussianNB

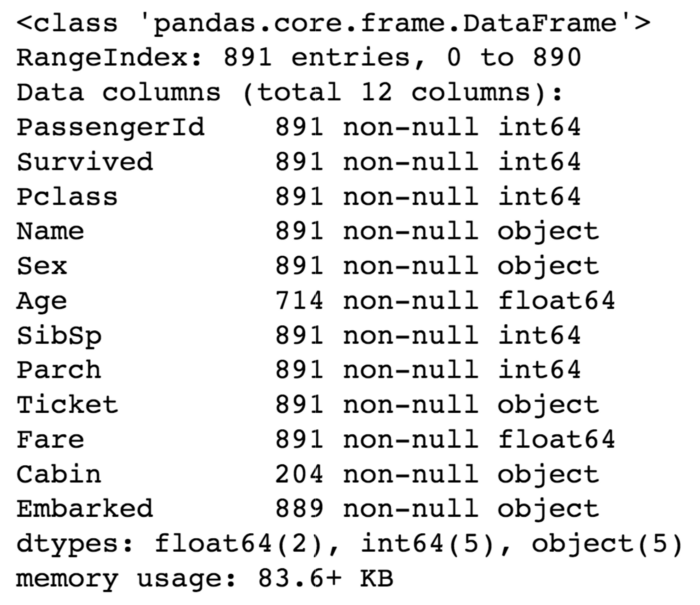
# **Getting the Data**

test\_df = pd.read\_csv("test.csv")

train\_df = pd.read\_csv("train.csv")

# **Data Exploration/Analysis**

train\_df.info()



**The training-set has 891 examples and 11 features + the target variable (survived)**. 2 of the features are floats, 5 are integers and 5 are objects. The features are listed below with a short description:

survival:    Survival

PassengerId: Unique Id of a passenger.

pclass:    Ticket class

sex:    Sex

Age:    Age in years

sibsp:    # of siblings / spouses aboard the Titanic

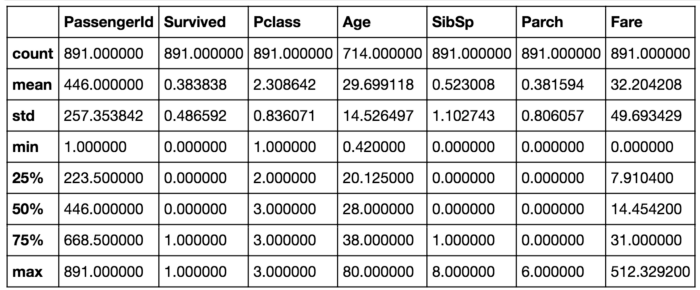
parch:    # of parents / children aboard the Titanic

ticket:    Ticket number

fare:    Passenger fare

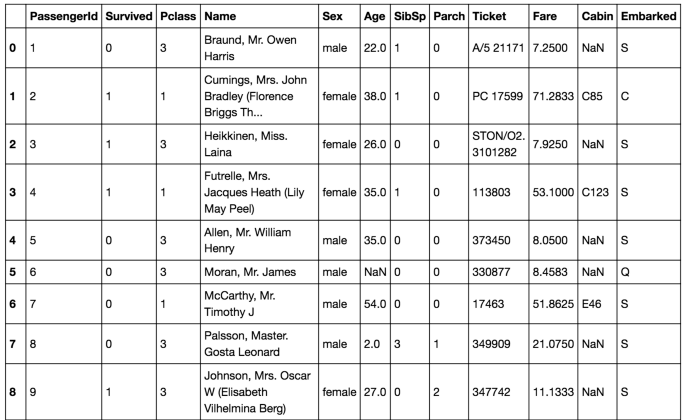
cabin:    Cabin number

embarked:    Port of Embarkationtrain\_df.describe()



From the above data it’s clear that about 38% survived the disaster. It’s also visible that age ranges from .4 to 80.

train\_df.head(8)



There are a few things that need to be taken into consideration from the above table .Lot of features need to be converted into numeric so that machine language can process them later.. Many features which have wide ranges need to be bought into the same scale.Some features which contain missing values also need to be handled.

**Let’s take a more detailed look at what data is actually missing:**

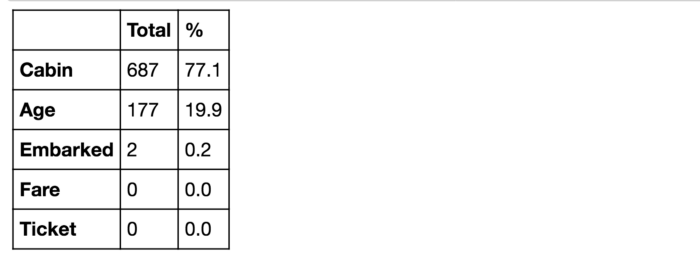
total = train\_df.isnull().sum().sort\_values(ascending=**False**)

percent\_1 = train\_df.isnull().sum()/train\_df.isnull().count()\*100

percent\_2 = (round(percent\_1, 1)).sort\_values(ascending=**False**)

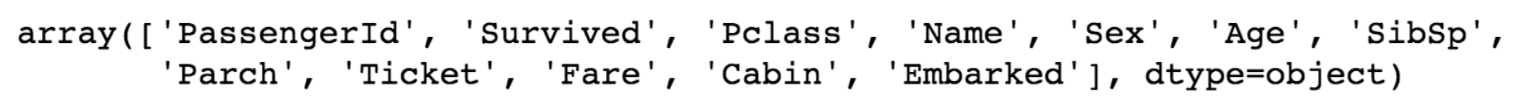
missing\_data = pd.concat([total, percent\_2], axis=1, keys=['Total', '%'])

missing\_data.head(5)



The two missing values of the embarked feature can be easily filled.Where as it will be more difficult to handle features like age which has 177 missing values.

train\_df.columns.values



Above you can see the 11 features + the target variable (survived). **What features could contribute to a high survival rate ?**

The above set makes sense if everything except ‘PassengerId’, ‘Ticket’ and ‘Name’ would be correlated with a high survival rate.

**1. Age and Sex:**

survived = 'survived'

not\_survived = 'not survived'

fig, axes = plt.subplots(nrows=1, ncols=2,figsize=(10, 4))

women = train\_df[train\_df['Sex']=='female']

men = train\_df[train\_df['Sex']=='male']

ax = sns.distplot(women[women['Survived']==1].Age.dropna(), bins=18, label = survived, ax = axes[0], kde =**False**)

ax = sns.distplot(women[women['Survived']==0].Age.dropna(), bins=40, label = not\_survived, ax = axes[0], kde =**False**)

ax.legend()

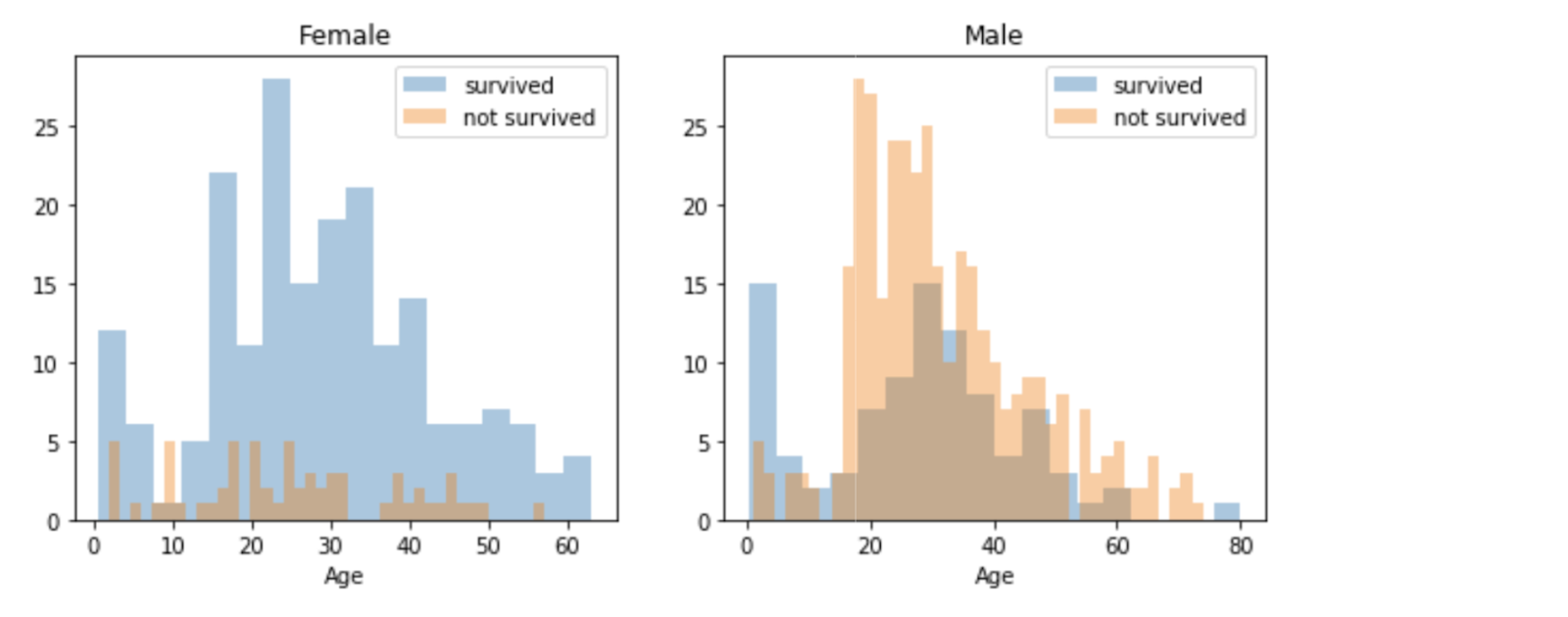
ax.set\_title('Female')

ax = sns.distplot(men[men['Survived']==1].Age.dropna(), bins=18, label = survived, ax = axes[1], kde = **False**)

ax = sns.distplot(men[men['Survived']==0].Age.dropna(), bins=40, label = not\_survived, ax = axes[1], kde = **False**)

ax.legend()

\_ = ax.set\_title('Male')



It’s clear that men have a high probability of survival when they are between 18 and 30 years old, which is partially true for women. For women it ranges between 14 and 40.

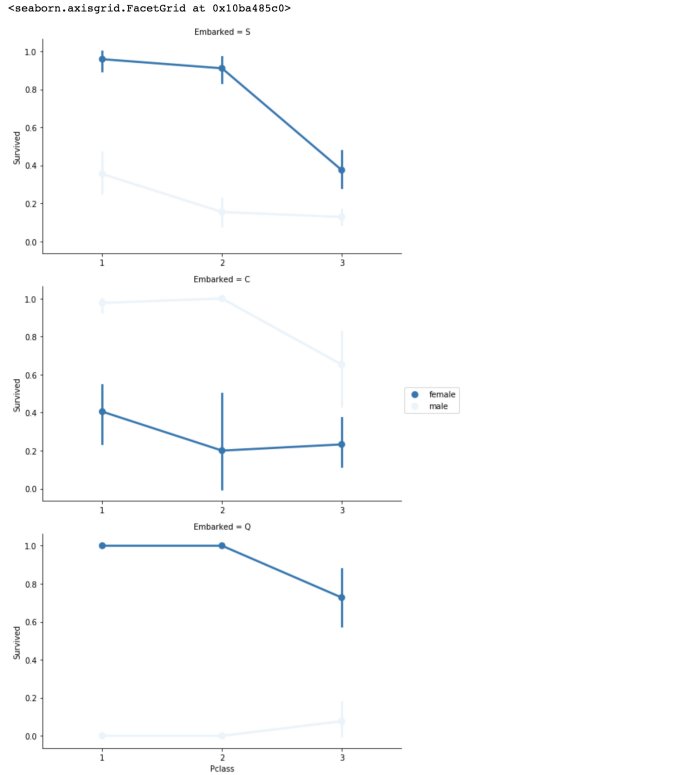
The probability of survival is very low between the age of 5 and 18 for men, which isn’t the case for women. Another thing to note is that infants also have a little bit higher probability of survival.

**3. Embarked, Pclass and Sex:**

FacetGrid = sns.FacetGrid(train\_df, row='Embarked', size=4.5, aspect=1.6)

FacetGrid.map(sns.pointplot, 'Pclass', 'Survived', 'Sex', palette=**None**,  order=**None**, hue\_order=**None** )

FacetGrid.add\_legend()



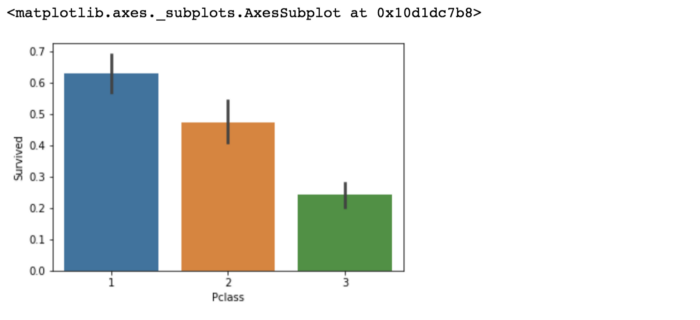
Embarked seems to be correlated with survival, depending on the gender.

Higher chance of survival is for women on port Q and S. If they are at port C inverse is true. Men have a high survival probability if they are on port C, but a low probability if they are on port Q or S.

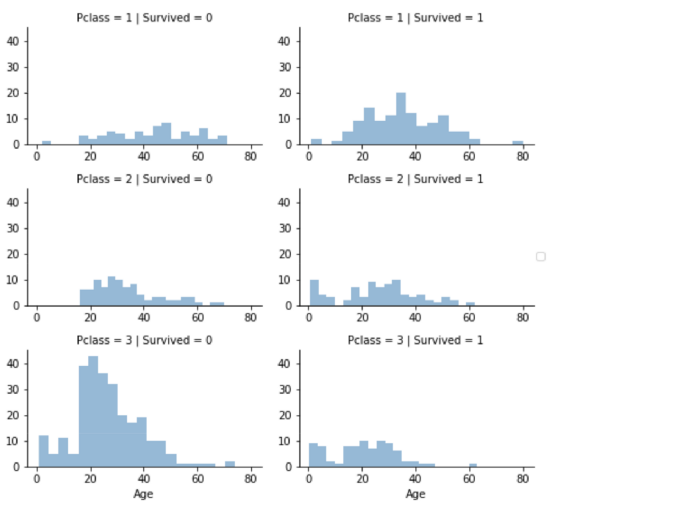
Pclass also seems to be correlated with survival. We will generate another plot of it below.

**4. Pclass:**

sns.barplot(x='Pclass', y='Survived', data=train\_df)



Here we see clearly, that Pclass is contributing to a persons chance of survival, especially if this person is in class 1. We will create another pclass plot below.



The plot above confirms our assumption about pclass 1, but we can also spot a high probability that a person in pclass 3 will not survive.

**5. SibSp and Parch:**

SibSp and Parch would make more sense as a combined feature that shows the total number of relatives a person has on the Titanic. I will create it below and also a feature that shows if someone is not alone.

data = [train\_df, test\_df]

**for** dataset **in** data:

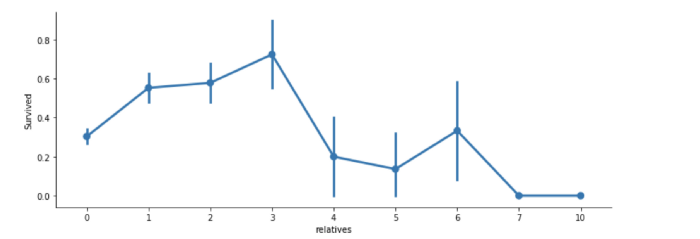
   dataset['relatives'] = dataset['SibSp'] + dataset['Parch']

   dataset.loc[dataset['relatives'] > 0, 'not\_alone'] = 0

   dataset.loc[dataset['relatives'] == 0, 'not\_alone'] = 1

   dataset['not\_alone'] = dataset['not\_alone'].astype(int)train\_df['not\_alone'].value\_counts()





# Data Preprocessing

First, I will drop ‘PassengerId’ from the train set, because it does not contribute to a person's survival probability. I will not drop it from the test set, since it is required there for the submission.

train\_df = train\_df.drop(['PassengerId'], axis=1)

## Missing Data:

Cabin:

We have to deal with Cabin (687), Embarked (2) and Age (177). A cabin number looks like ‘C123’ and the letter refers to the deck. Therefore we’re going to extract these and create a new feature, that contains a person's deck. Afterwards we will convert the feature into a numeric variable. The missing values will be converted to zero. In the picture below you can see the actual decks of the titanic, ranging from A to G.

import re

deck = {"A": 1, "B": 2, "C": 3, "D": 4, "E": 5, "F": 6, "G": 7, "U": 8}

data = [train\_df, test\_df]

for dataset in data:

   dataset['Cabin'] = dataset['Cabin'].fillna("U0")

   dataset['Deck'] = dataset['Cabin'].map(lambda x: re.compile("([a-zA-Z]+)").search(x).group())

   dataset['Deck'] = dataset['Deck'].map(deck)

   dataset['Deck'] = dataset['Deck'].fillna(0)

   dataset['Deck'] = dataset['Deck'].astype(int)*# we can now drop the cabin feature*

train\_df = train\_df.drop(['Cabin'], axis=1)

test\_df = test\_df.drop(['Cabin'], axis=1)

Age:

Now we can tackle the issue with the age features missing values. I will create an array that contains random numbers, which are computed based on the mean age value in regards to the standard deviation and is\_null.

data = [train\_df, test\_df]

for dataset in data:

   mean = train\_df["Age"].mean()

   std = test\_df["Age"].std()

   is\_null = dataset["Age"].isnull().sum()

*# compute random numbers between the mean, std and is\_null*

   rand\_age = np.random.randint(mean - std, mean + std, size = is\_null)

*# fill NaN values in Age column with random values generated*

   age\_slice = dataset["Age"].copy()

   age\_slice[np.isnan(age\_slice)] = rand\_age

   dataset["Age"] = age\_slice

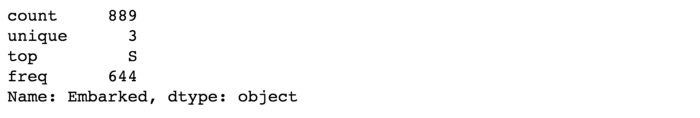
   dataset["Age"] = train\_df["Age"].astype(int)train\_df["Age"].isnull().sum()



Embarked:

Since the Embarked feature has only 2 missing values, we will just fill these with the most common one.

train\_df['Embarked'].describe()



Fare:

Converting “Fare” from float to int64, using the “astype()” function pandas provides:

data = [train\_df, test\_df]

for dataset in data:

   dataset['Fare'] = dataset['Fare'].fillna(0)

   dataset['Fare'] = dataset['Fare'].astype(int)

Name:

We will use the Name feature to extract the Titles from the Name, so that we can build a new feature out of that.

data = [train\_df, test\_df]

titles = {"Mr": 1, "Miss": 2, "Mrs": 3, "Master": 4, "Rare": 5}

for dataset in data:

*# extract titles*

   dataset['Title'] = dataset.Name.str.extract(' ([A-Za-z]+)\.', expand=False)

*# replace titles with a more common title or as Rare*

   dataset['Title'] = dataset['Title'].replace(['Lady', 'Countess','Capt', 'Col','Don', 'Dr',\

                                           'Major', 'Rev', 'Sir', 'Jonkheer', 'Dona'], 'Rare')

   dataset['Title'] = dataset['Title'].replace('Mlle', 'Miss')

   dataset['Title'] = dataset['Title'].replace('Ms', 'Miss')

   dataset['Title'] = dataset['Title'].replace('Mme', 'Mrs')

*# convert titles into numbers*

   dataset['Title'] = dataset['Title'].map(titles)

*# filling NaN with 0, to get safe*

   dataset['Title'] = dataset['Title'].fillna(0)train\_df = train\_df.drop(['Name'], axis=1)

test\_df = test\_df.drop(['Name'], axis=1)

Sex:

Convert ‘Sex’ feature into numeric.

genders = {"male": 0, "female": 1}

data = [train\_df, test\_df]

for dataset in data:

   dataset['Sex'] = dataset['Sex'].map(genders)

Ticket:

train\_df['Ticket'].describe()

Since the Ticket attribute has 681 unique tickets, it will be a bit difficult to convert them into useful categories. So we will drop it from the dataset.

train\_df = train\_df.drop(['Ticket'], axis=1)

test\_df = test\_df.drop(['Ticket'], axis=1)

Embarked:

Convert the ‘Embarked’ feature into numeric.

ports = {"S": 0, "C": 1, "Q": 2}

data = [train\_df, test\_df]

for dataset in data:

   dataset['Embarked'] = dataset['Embarked'].map(ports)

# Creating Categories:

We will now create categories within the following features:

Age:

The ‘age’ feature is to be converted now. For that it will be converted from float to integer first. After that the new ‘AgeGroup” variable will be created, by categorizing every age into a group. data = [train\_df, test\_df]

for dataset in data:

   dataset['Age'] = dataset['Age'].astype(int)

   dataset.loc[ dataset['Age'] <= 11, 'Age'] = 0

   dataset.loc[(dataset['Age'] > 11) & (dataset['Age'] <= 18), 'Age'] = 1

   dataset.loc[(dataset['Age'] > 18) & (dataset['Age'] <= 22), 'Age'] = 2

   dataset.loc[(dataset['Age'] > 22) & (dataset['Age'] <= 27), 'Age'] = 3

   dataset.loc[(dataset['Age'] > 27) & (dataset['Age'] <= 33), 'Age'] = 4

   dataset.loc[(dataset['Age'] > 33) & (dataset['Age'] <= 40), 'Age'] = 5

   dataset.loc[(dataset['Age'] > 40) & (dataset['Age'] <= 66), 'Age'] = 6

   dataset.loc[ dataset['Age'] > 66, 'Age'] = 6

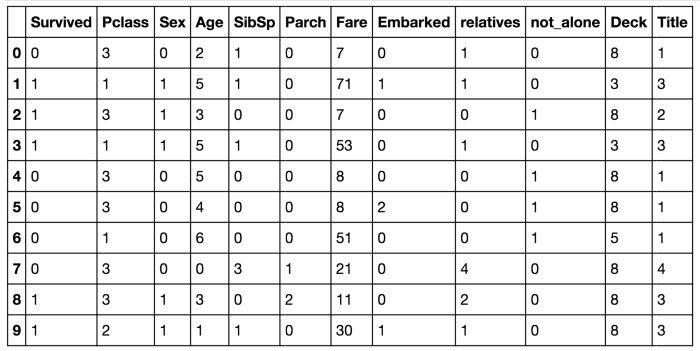
*# let's see how it's distributed* train\_df['Age'].value\_counts()



Fare:

For the ‘Fare’ feature, we need to do the same as with the ‘Age’ feature. But it isn’t that easy, because if we cut the range of the fare values into a few equally big categories, 80% of the values would fall into the first category. Fortunately, we can use sklearn “qcut()” function, that we can use to see how we can form the categories.

train\_df.head(10)



# Building Machine Learning Models

 Here several Machine Learning models will be trained and  their results will be compared. Since dataset does not provide labels for their testing-set, we need to use the predictions on the training set to compare the algorithms with each other. Later on, we will use cross validation.

X\_train = train\_df.drop("Survived", axis=1)

Y\_train = train\_df["Survived"]

X\_test  = test\_df.drop("PassengerId", axis=1).copy()

Stochastic Gradient Descent (SGD):

sgd = linear\_model.SGDClassifier(max\_iter=5, tol=None)

sgd.fit(X\_train, Y\_train)

Y\_pred = sgd.predict(X\_test)

sgd.score(X\_train, Y\_train)

acc\_sgd = round(sgd.score(X\_train, Y\_train) \* 100, 2)

Random Forest:

random\_forest = RandomForestClassifier(n\_estimators=100)

random\_forest.fit(X\_train, Y\_train)

Y\_prediction = random\_forest.predict(X\_test)

random\_forest.score(X\_train, Y\_train)

acc\_random\_forest = round(random\_forest.score(X\_train, Y\_train) \* 100, 2)

Logistic Regression:

logreg = LogisticRegression()

logreg.fit(X\_train, Y\_train)

Y\_pred = logreg.predict(X\_test)

acc\_log = round(logreg.score(X\_train, Y\_train) \* 100, 2)

K Nearest Neighbor:

*# KNN* knn = KNeighborsClassifier(n\_neighbors = 3) knn.fit(X\_train, Y\_train)  Y\_pred = knn.predict(X\_test)  acc\_knn = round(knn.score(X\_train, Y\_train) \* 100, 2)

Gaussian Naive Bayes:

gaussian = GaussianNB() gaussian.fit(X\_train, Y\_train)  Y\_pred = gaussian.predict(X\_test)  acc\_gaussian = round(gaussian.score(X\_train, Y\_train) \* 100, 2)

Perceptron:

perceptron = Perceptron(max\_iter=5)

perceptron.fit(X\_train, Y\_train)

Y\_pred = perceptron.predict(X\_test)

acc\_perceptron = round(perceptron.score(X\_train, Y\_train) \* 100, 2)

Linear Support Vector Machine:

linear\_svc = LinearSVC()

linear\_svc.fit(X\_train, Y\_train)

Y\_pred = linear\_svc.predict(X\_test)

acc\_linear\_svc = round(linear\_svc.score(X\_train, Y\_train) \* 100, 2)

Decision Tree

decision\_tree = DecisionTreeClassifier() decision\_tree.fit(X\_train, Y\_train)  Y\_pred = decision\_tree.predict(X\_test)  acc\_decision\_tree = round(decision\_tree.score(X\_train, Y\_train) \* 100, 2)

# Which is the best Model ?

results = pd.DataFrame({

   'Model': ['Support Vector Machines', 'KNN', 'Logistic Regression',

             'Random Forest', 'Naive Bayes', 'Perceptron',

             'Stochastic Gradient Decent',

             'Decision Tree'],

   'Score': [acc\_linear\_svc, acc\_knn, acc\_log,

             acc\_random\_forest, acc\_gaussian, acc\_perceptron,

             acc\_sgd, acc\_decision\_tree]})

result\_df = results.sort\_values(by='Score', ascending=False)

result\_df = result\_df.set\_index('Score')

result\_df.head(9)



# Random Forest

## What is Random Forest ?

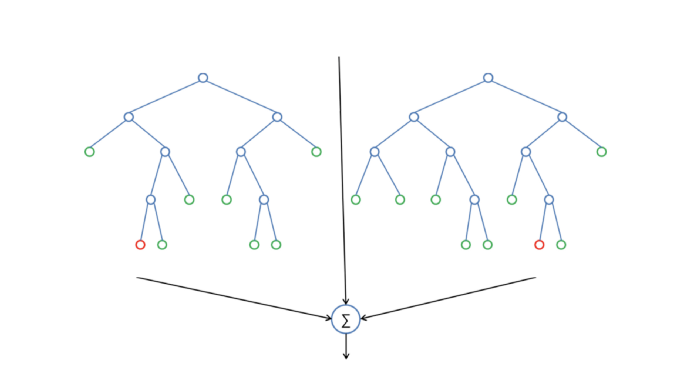
It is a supervised learning algorithm. As its name suggests, it creates a forest and makes it somehow random. The forest built is an ensemble of Decision Trees, most of the time trained with the “bagging” method. The general idea of the bagging method is that a combination of learning models increases the overall result.

Simply, Random forest builds multiple decision trees which are merged together to get a more accurate and stable prediction.

One of its noteworthy advantages is that it can be used for both classification and regression problems, which constitute the majority of current machine learning systems. Barring a few exceptions, a random-forest classifier has all the hyperparameters of a decision-tree classifier and also all the hyperparameters of a bagging classifier, to control the ensemble itself.

The random-forest algorithm brings extra randomness into the model, when it is growing the trees. Instead of searching for the best feature while splitting a node, it searches for the best feature among a random subset of features. This process creates a wide diversity, which generally results in a better model. Therefore when you are growing a tree in a random forest, only a random subset of the features is considered for splitting a node. You can even make trees more random, by using random thresholds on top of it, for each feature rather than searching for the best possible thresholds (like a normal decision tree does).

Below you can see how a random forest would look like with two trees:



## Feature Importance

Another great quality of random forest is that they make it very easy to measure the relative importance of each feature. Sklearn measures a features importance by looking at how much the tree nodes that use that feature, reduce impurity on average (across all trees in the forest). It computes this score automatically for each feature after training and scales the results so that the sum of all importances is equal to 1. We will access this below:

importances = pd.DataFrame({'feature':X\_train.columns,'importance':np.round(random\_forest.feature\_importances\_,3)})

importances = importances.sort\_values('importance',ascending=False).set\_index('feature')importances.head(15)

## Conclusion:

not\_alone and Parch doesn’t play a significant role in our random forest classifiers prediction process. Because of that I will drop them from the dataset and train the classifier again. We could also remove more or less features, but this would need a more detailed investigation of the features effect on our model. But I think it’s just fine to remove only Alone and Parch.

train\_df  = train\_df.drop("not\_alone", axis=1)

test\_df  = test\_df.drop("not\_alone", axis=1)

train\_df  = train\_df.drop("Parch", axis=1)

test\_df  = test\_df.drop("Parch", axis=1)