from KaggleAux import predict as ka # see github.com/agconti/kaggleaux for more details

# Create a new array with the added features: features\_two

features\_two = train[["Pclass","Age","Sex","Fare", "SibSp", "Parch", "Embarked"]].values

#Control overfitting by setting "max\_depth" to 10 and "min\_samples\_split" to 5 : my\_tree\_two

max\_depth = 10

min\_samples\_split = 5

my\_tree\_two = tree.DecisionTreeClassifier(max\_depth = 10, min\_samples\_split = 5, random\_state = 1)

my\_tree\_two = my\_tree\_two.fit(features\_two, target)

#Print the score of the new decison tree

print(my\_tree\_two.score(features\_two, target))

# Create train\_two with the newly defined feature

train\_two = train.copy()

train\_two["family\_size"] = train["SibSp"] + train["Parch"] + 1

# Create a new feature set and add the new feature

features\_three = train\_two[["Pclass", "Sex", "Age", "Fare", "SibSp", "Parch", "family\_size"]].values

# Define the tree classifier, then fit the model

my\_tree\_three = tree.DecisionTreeClassifier()

my\_tree\_three = my\_tree\_three.fit(features\_three, target)

# Print the score of this decision tree

print(my\_tree\_three.score(features\_three, target))

# Import the `RandomForestClassifier`

from sklearn.ensemble import RandomForestClassifier

# We want the Pclass, Age, Sex, Fare,SibSp, Parch, and Embarked variables

features\_forest = train[["Pclass", "Age", "Sex", "Fare", "SibSp", "Parch", "Embarked"]].values

# Building and fitting my\_forest

forest = RandomForestClassifier(max\_depth = 10, min\_samples\_split=2, n\_estimators = 100, random\_state = 1)

my\_forest = forest.fit(features\_forest, target)

# Print the score of the fitted random forest

print(my\_forest.score(features\_forest, target))

# Compute predictions on our test set features then print the length of the prediction vector

test\_features = test[["Pclass", "Age", "Sex", "Fare", "SibSp", "Parch", "Embarked"]].values

pred\_forest = my\_forest.predict(test\_features)

print(len(pred\_forest))

#Request and print the `.feature\_importances\_` attribute

print(my\_tree\_two.feature\_importances\_)

print(my\_forest.feature\_importances\_)

#Compute and print the mean accuracy score for both models

print(my\_tree\_two.score(features\_two, target))

print(my\_forest.score(features\_forest, target))

--

def cleaneddf(no\_bins=0):

#you'll want to tweak this to conform with your computer's file system

trainpath = 'C:/Documents and Settings/DIGIT/My Documents/Google Drive/Blogs/triangleinequality/Titanic/rawtrain.csv'

testpath = 'C:/Documents and Settings/DIGIT/My Documents/Google Drive/Blogs/triangleinequality/Titanic/rawtest.csv'

traindf = pd.read\_csv(trainpath)

testdf = pd.read\_csv(testpath)

#discretise fare

if no\_bins==0:

return [cleandf(traindf), cleandf(testdf)]

traindf=cleandf(traindf)

testdf=cleandf(testdf)

bins\_and\_binned\_fare = pd.qcut(traindf.Fare, no\_bins, retbins=True)

bins=bins\_and\_binned\_fare[1]

traindf.Fare = bins\_and\_binned\_fare[0]

testdf.Fare = pd.cut(testdf.Fare, bins)

#discretise age

bins\_and\_binned\_age = pd.qcut(traindf.Age, no\_bins, retbins=True)

bins=bins\_and\_binned\_age[1]

traindf.Age = bins\_and\_binned\_age[0]

testdf.Age = pd.cut(testdf.Age, bins)

#create a submission file for kaggle

predictiondf = pd.DataFrame(testdf['PassengerId'])

predictiondf['Survived']=[0 for x in range(len(testdf))]

predictiondf.to\_csv('C:/Documents and Settings/DIGIT/My Documents/Google Drive/Blogs/triangleinequality/Titanic/prediction.csv',

index=False)

return [traindf, testdf]

--

import pandas as pd

import numpy as np

from scipy.stats import mode

def cleandf(df):

#cleaning fare column

df.Fare = df.Fare.map(lambda x: np.nan if x==0 else x)

classmeans = df.pivot\_table('Fare', rows='Pclass', aggfunc='mean')

df.Fare = df[['Fare', 'Pclass']].apply(lambda x: classmeans[x['Pclass']] if pd.isnull(x['Fare']) else x['Fare'], axis=1 )

#cleaning the age column

meanAge=np.mean(df.Age)

df.Age=df.Age.fillna(meanAge)

#cleaning the embarked column

df.Cabin = df.Cabin.fillna('Unknown')

modeEmbarked = mode(df.Embarked)[0][0]

df.embarked = df.Embarked.fillna(modeEmbarked)

return df

def cleaneddf(no\_bins=0):

#you'll want to tweak this to conform with your computer's file system

trainpath = 'C:/Documents and Settings/DIGIT/My Documents/Google Drive/Blogs/triangleinequality/Titanic/rawtrain.csv'

testpath = 'C:/Documents and Settings/DIGIT/My Documents/Google Drive/Blogs/triangleinequality/Titanic/rawtest.csv'

traindf = pd.read\_csv(trainpath)

testdf = pd.read\_csv(testpath)

#discretise fare

if no\_bins==0:

return [cleandf(traindf), cleandf(testdf)]

traindf=cleandf(traindf)

testdf=cleandf(testdf)

bins\_and\_binned\_fare = pd.qcut(traindf.Fare, no\_bins, retbins=True)

bins=bins\_and\_binned\_fare[1]

traindf.Fare = bins\_and\_binned\_fare[0]

testdf.Fare = pd.cut(testdf.Fare, bins)

#discretise age

bins\_and\_binned\_age = pd.qcut(traindf.Age, no\_bins, retbins=True)

bins=bins\_and\_binned\_age[1]

traindf.Age = bins\_and\_binned\_age[0]

testdf.Age = pd.cut(testdf.Age, bins)

#create a submission file for kaggle

predictiondf = pd.DataFrame(testdf['PassengerId'])

predictiondf['Survived']=[0 for x in range(len(testdf))]

predictiondf.to\_csv('C:/Documents and Settings/DIGIT/My Documents/Google Drive/Blogs/triangleinequality/Titanic/prediction.csv',

index=False)

return [traindf, testdf]

--

# -\*- coding: utf-8 -\*-

import numpy as np

import igraph as ig

class BasicNode: #should be interfaced to from a graph object

def \_\_init\_\_(self, content=None, labels=None):

self.content = content

self.incident\_edges =set([])

self.incident\_outward\_edges=set([])

self.incident\_inward\_edges=set([])

if labels is None:

self.labels=set([])

else:

self.labels=labels

def add\_edge(self,Edge):

if self in Edge.ends:

self.incident\_edges.add(Edge)

if Edge.source()==self:

self.incident\_outward\_edges.add(Edge)

else:

self.incident\_inward\_edges.add(Edge)

else:

print 'Cannot add edge to vertex, vertex not in ends.'

def remove\_edge(self,Edge):

self.incident\_edges.discard(Edge)

def get\_neighbors(self):

neighbors = [Edge.ends for Edge in self.incident\_edges]

unique\_neighbors = list(set(reduce(lambda x,y: x+y, neighbors)))

if [self, self] not in neighbors: #checks for a loop

unique\_neighbors.remove(self)

return unique\_neighbors

def get\_targets(self):

#targets = set([Edge.target() for Edge in self.incident\_outward\_edges])

targets = set(map(lambda x: x.target(), self.incident\_outward\_edges))

return targets

def get\_sources(self):

#sources = set([Edge.source() for Edge in self.incident\_inward\_edges])

sources = set(map(lambda x: x.source(), self.incident\_inward\_edges))

return sources

def add\_label(self, label):

self.labels.add(label)

def remove\_label(self, label):

self.labels.discard(label)

class BasicEdge: #should be interfaced to from a graph object

def \_\_init\_\_(self, content=None, ends=[], labels=None):

self.content = content

self.ends = ends

if labels is None:

self.labels=set([])

else:

self.labels=labels

def source(self):

return self.ends[0]

def target(self):

return self.ends[1]

def add\_label(self, label):

self.labels.add(label)

def remove\_label(self, label):

self.labels.discard(label)

def update\_up(class\_method):

def inner(self, \*args, \*\*kwargs):

method\_name = class\_method.func\_name

class\_method(self, \*args, \*\*kwargs)

for Supergraph in self.supergraphs:

getattr(Supergraph, method\_name)(\*args, \*\*kwargs)

return inner

def update\_up\_down(class\_method):

def inner(self, \*args, \*\*kwargs):

method\_name = class\_method.func\_name

if class\_method(self, \*args, \*\*kwargs):

for Supergraph in self.supergraphs:

getattr(Supergraph, method\_name)(\*args, \*\*kwargs)

for Subgraph in self.subgraphs:

getattr(Subgraph, method\_name)(\*args, \*\*kwargs)

return inner

class Graph(object):

def \_\_init\_\_(self, subgraphs=None, supergraphs=None, vertices=None, edges=None, Vertex=BasicNode, Edge=BasicEdge):

if edges == None:

edges = []

self.edges=set(edges)

if vertices == None:

vertices = []

self.vertices=vertices

if subgraphs == None:

subgraphs=[]

self.subgraphs =set(subgraphs)

if supergraphs == None:

supergraphs =[]

self.supergraphs=set(supergraphs)

for Supergraph in supergraphs:

Supergraph.add\_subgraph(self)

for Subgraph in subgraphs:

Subgraph.add\_supergraph(self)

self.vertex\_dict = {}

self.edges\_dict={}

self.Vertex = Vertex

self.Edge = Edge

@update\_up

def create\_vertex(self):

self.vertices.append(self.Vertex())

def create\_vertices(self, no\_create):

for i in range(no\_create):

self.create\_vertex()

@update\_up

def add\_vertex(self, Vertex):

if Vertex in self.vertices:

return

self.vertices.append(Vertex)

@update\_up

def create\_edge(self, ends):

NewEdge=self.Edge(ends=ends)

self.edges.add(NewEdge)

for Vertex in ends:

Vertex.add\_edge(NewEdge)

@update\_up\_down

def remove\_edge(self, Edge):

if not Edge in self.edges:

return False

self.edges.discard(Edge)

return True

def get\_incident\_edges(self, Vertex):

incident\_edges = Vertex.incident\_edges & self.edges

return incident\_edges

@update\_up\_down

def remove\_vertex(self, Vertex):

if Vertex not in self.vertices:

return False

edges\_to\_remove = self.get\_incident\_edges(Vertex)

for Edge in edges\_to\_remove:

self.remove\_edge(Edge)

self.vertices.remove(Vertex)

return True

def get\_vertex\_neighbors(self, Vertex):

neighbors = (Vertex.get\_neighbors() & set(self.vertices))

return neighbors

def get\_degree(self, Vertex):

return len(self.get\_incident\_edges(Vertex))

def get\_number\_vertices(self):

return len(self.vertices)

def get\_number\_edges(self):

return len(self.edges)

def get\_adjacency\_matrix(self):

#adj\_list = [self.get\_adjacency\_list\_of\_vertex(Vertex) for Vertex in self.vertices]

adj\_list = map(lambda x: self.get\_adjacency\_list\_of\_vertex(x), self.vertices)

adj\_mat = np.array(adj\_list)

return adj\_mat

def get\_adjacency\_matrix\_as\_list(self):

return self.get\_adjacency\_matrix().tolist()

def set\_adjacency\_list(self, adj\_list):

self.vertices=[]

self.edges=[]

def add\_subgraph(self,Subgraph):

self.subgraphs.add(Subgraph)

def add\_supergraph(self, Supergraph):

self.supergraphs.add(Supergraph)

def is\_in(self,vertex\_or\_edge):

if (vertex\_or\_edge in self.edges) or (vertex\_or\_edge in self.vertices):

return True

else:

return False

def get\_incident\_outward\_edges(self,Vertex):

return (Vertex.incident\_outward\_edges & self.edges)

def get\_incident\_inward\_edges(self,Vertex):

return (Vertex.incident\_inward\_edges & self.edges)

def get\_vertex\_targets(self, Vertex):

targets = (Vertex.get\_targets() & set(self.vertices))

return targets

def get\_vertex\_sources(self, Vertex):

sources = (Vertex.get\_sources() & set(self.vertices))

return sources

def add\_vertex\_label(self, vertex, label):

self.vertex\_dict[label] = vertex

vertex.add\_label(label)

def get\_vertex(self,label):

if label in self.vertex\_dict.keys():

return self.vertex\_dict[label]

else:

return None

def get\_vertex\_label(self, vertex):

labels = vertex.get\_labels()

labels = labels & self.vertex\_dict.keys()

labels = filter(lambda x: self.get\_vertex[x]==vertex,labels)

def remove\_vertex\_label(self, label):

vertex=self.vertex\_dict.pop(label, 'Not Found')

if vertex == 'Not Found':

return

else:

vertex.remove\_label(label)

def add\_edge\_label(self, edge, label):

self.edge\_dict[label] = edge

edge.add\_label(label)

def get\_edge(self,label):

if label in self.edge\_dict.keys():

return self.edge\_dict[label]

else:

return None

def get\_edge\_label(self, edge):

labels = edge.get\_labels()

labels = labels & self.edge\_dict.keys()

labels = filter(lambda x: self.get\_edge[x]==edge,labels)

def remove\_edge\_label(self, label):

edge=self.edge\_dict.pop(label, 'Not Found')

if edge == 'Not Found':

return

else:

edge.remove\_label(label)

class UnDirGraph(Graph, object):

def get\_adjacency\_list\_of\_vertex(self, Vertex):

N = self.get\_number\_vertices()

adj\_list= [0 for x in range(N)]

incident\_edges = self.get\_incident\_edges(Vertex)

for Edge in incident\_edges:

ends = Edge.ends

if ends[0] != Vertex:

index = self.vertices.index(ends[0])

else:

index = self.vertices.index(ends[1])

adj\_list[index] += 1

return adj\_list

def set\_adjacency\_matrix(self, adj\_mat):

shape = np.shape(adj\_mat)

if shape[0] != shape[1]:

print 'Wrong shape, expecting square matrix.'

return

n = shape[0]

self.vertices=[]

self.edges=[]

self.create\_vertices(n)

for row in range(n):

Source = self.vertices[row]

for col in range(row+1):

no\_edges = adj\_mat[row, col]

Target = self.vertices[col]

for Edge in range(no\_edges):

self.create\_edge(ends=[Source, Target])

def plot(self):

A = self.get\_adjacency\_matrix\_as\_list()

convert\_to\_igraph = ig.Graph.Adjacency(A, mode='undirected')

ig.plot(convert\_to\_igraph)

class DirGraph(Graph):

def get\_incident\_outward\_edges(self,Vertex):

return (Vertex.incident\_outward\_edges & self.edges)

def get\_incident\_inward\_edges(self,Vertex):

return (Vertex.incident\_inward\_edges & self.edges)

def get\_adjacency\_list\_of\_vertex(self, Vertex):

N = self.get\_number\_vertices()

adj\_list= [0 for x in range(N)]

incident\_edges = self.get\_incident\_outward\_edges(Vertex)

for Edge in incident\_edges:

target = Edge.target()

index = self.vertices.index(target)

adj\_list[index] += 1

return adj\_list

def set\_adjacency\_matrix(self, adj\_mat):

shape = np.shape(adj\_mat)

if shape[0] != shape[1]:

print 'Wrong shape, expecting square matrix.'

return

n = shape[0]

self.vertices=[]

self.edges=[]

self.create\_vertices(n)

for row, col in range(n):

no\_edges = adj\_mat[row, col]

Source = self.vertices[row]

Target = self.vertices[col]

for Edge in range(no\_edges):

self.create\_edge(ends=[Source, Target])

def get\_vertex\_targets(self, Vertex):

targets = (Vertex.get\_targets() & set(self.vertices))

return targets

def get\_vertex\_sources(self, Vertex):

sources = (Vertex.get\_sources() & set(self.vertices))

return sources

def plot(self):

A = self.get\_adjacency\_matrix\_as\_list()

convert\_to\_igraph = ig.Graph.Adjacency(A)

ig.plot(convert\_to\_igraph)

#This is a wrapper to a class definition, deciding whether to inherit

#from DirGraph or UnDirGraph at runtime. It can be initialised by

#number of vertices or the number of edges.

def return\_linear\_class(directed=False):

if directed:

base=DirGraph

else:

base=UnDirGraph

class Linear(base, object):

def \_\_init\_\_(self, number\_vertices=0, number\_edges=0, \*\*kwargs):

super(Linear, self).\_\_init\_\_(\*\*kwargs)

self.linear\_generate(number\_vertices, number\_edges)

def linear\_generate(self,number\_vertices, number\_edges):

if (not number\_edges ==0) and (not number\_vertices==0):

if not number\_vertices == number\_edges+1:

print 'Number of edges and vertices incompatible!'

return

else:

self.number\_vertices=number\_vertices

elif not number\_edges==0:

self.number\_vertices = number\_edges +1

else:

self.number\_vertices = number\_vertices

self.create\_vertices(self.number\_vertices)

for index in range(self.number\_vertices -1):

Source = self.vertices[index]

Target = self.vertices[index+1]

self.create\_edge([Source, Target])

return Linear

#instantiates the Linear class

def create\_linear(directed=False, number\_vertices=0, number\_edges=0,\*\*kwargs):

linear = return\_linear\_class(directed)(number\_vertices, number\_edges,\*\*kwargs)

return linear

#Class definition wrapper to dynamically inherti from DirGraph or UnDirGraph.

#Also has a composition from Linear, to create a cycle it joins the ends

#of a linear graph.

def return\_cycle\_class(directed=False):

if directed:

base = DirGraph

else:

base = UnDirGraph

class Cycle(base, object):

def \_\_init\_\_(self, number\_vertices=0, number\_edges=0, \*\*kwargs):

super(Cycle, self).\_\_init\_\_(\*\*kwargs)

if (not number\_edges==0) and (not number\_vertices==0):

if not number\_edges == number\_vertices:

print 'Numbers of edges and vertices incompatible!'

return

elif not number\_edges ==0:

number\_vertices = number\_edges

Linear\_part = create\_linear()

Linear\_part.linear\_generate(number\_vertices, number\_edges-1)

self.vertices = Linear\_part.vertices

self.edges = Linear\_part.edges

self.cycle\_generate(number\_vertices)

def cycle\_generate(self, number\_vertices):

Target = self.vertices[0]

Source = self.vertices[number\_vertices-1]

self.create\_edge(ends=[Source, Target])

return Cycle

def create\_cycle(directed=False, number\_vertices=0, number\_edges=0, \*\*kwargs):

cycle = return\_cycle\_class(directed)(number\_vertices, number\_edges, \*\*kwargs)

return cycle

class Complete(UnDirGraph, object):

def \_\_init\_\_(self,number\_vertices=0, \*\*kwargs):

super(Complete, self).\_\_init\_\_(\*\*kwargs)

self.create\_vertices(no\_create=number\_vertices)

ends=[]

for Source in self.vertices:

for Target in self.vertices:

if [Source,Target] not in ends:

if not Source == Target:

self.create\_edge(ends=[Source,Target])

ends.append([Source,Target])

ends.append([Target, Source])

def return\_tree\_class(directed=False):

if directed:

base = DirGraph

else:

base = UnDirGraph

class Tree(base, object):

def \_\_init\_\_(self, \*\*kwargs):

super(Tree, self).\_\_init\_\_(\*\*kwargs)

self.leaves=set([])

self.find\_leaves()

def is\_leaf(self, vertex):

if self.get\_degree(vertex) == 1:

return True

elif self.get\_number\_vertices() ==1:

return True

else:

return False

def set\_root(self, vertex):

if vertex in self.vertices:

self.remove\_vertex\_label('Root')

self.add\_vertex\_label(vertex, label='Root')

def get\_root(self):

return self.get\_vertex('Root')

def find\_leaves(self):

self.leaves = set(filter(self.is\_leaf,self.vertices))

return [leaf for leaf in self.leaves]

return Tree

def create\_tree(directed=False, \*\*kwargs):

tree = return\_tree\_class(directed)(\*\*kwargs)

return tree

def return\_nary\_tree\_class(directed=False):

base = return\_tree\_class(directed)

class NaryRootedTree(base,object):

def \_\_init\_\_(self, N=0, \*\*kwargs):

super(NaryRootedTree, self).\_\_init\_\_(\*\*kwargs)

self.N = N

def split\_vertex(self, vertex):

if vertex in self.leaves:

children = [self.Vertex() for i in range(self.N)]

vertex.children = children

self.leaves.discard(vertex)

for Child in children:

self.add\_vertex(Child)

self.leaves.add(Child)

Child.parent=vertex

self.create\_edge(ends=[vertex, Child])

def fuse\_vertex(self, vertex):

self.leaves.add(vertex)

try:

children=vertex.children

except AttributeError:

return

if children==None:

return

for child in children:

self.fuse\_vertex(child)

self.leaves.discard(child)

self.remove\_vertex(child)

child.parent = None

vertex.children = None

def create\_full\_n\_level(self, n):

self.vertices =[]

self.edges =set([])

self.create\_vertex()

self.set\_root(self.vertices[0])

for level in range(n):

leaves=self.find\_leaves()

for Leaf in leaves:

self.split\_vertex(Leaf)

def get\_descendants(self, node, desc=set({})):

try:

children=node.children

except AttributeError:

node.children=None

if children != None:

for child in children:

desc = desc.union(set({child}))

desc= desc.union(self.get\_descendants(child))

return desc

else:

return desc

return NaryRootedTree

def create\_nary\_tree(directed=False, N=0,\*\*kwargs):

nary\_tree = return\_nary\_tree\_class(directed)(N, \*\*kwargs)

return nary\_tree

--

# -\*- coding: utf-8 -\*-

import random

import ticart

import pandas as pd

import cleantitanic as ct

def cross\_validate(no\_folds, data, resample=False):

rows = list(data.index)

random.shuffle(rows)

N=len(data)

len\_fold = int(N/no\_folds)

start=0

for i in range(no\_folds):

if i==no\_folds-1:

stop =N

else:

stop = start +len\_fold

test = data.ix[rows[start:stop]]

train = data.ix[rows[:start]+rows[stop:]]

if resample:

train\_len=start+N-stop

no\_resamples = N-train\_len

train\_rows = list(train.index)

random\_extra\_rows =[random.choice(train\_rows) for row in range(no\_resamples)]

train\_rows = train\_rows+random\_extra\_rows

train=train.ix[train\_rows]

yield {'test':test, 'train':train}

start=stop

df=ct.cleaneddf(no\_bins=10)[0]

df2=ct.cleaneddf(no\_bins=10)[1]

df=df[['Survived', 'Pclass', 'Sex', 'Age', 'SibSp', 'Parch',

'Fare', 'Embarked']]

data\_type\_dict={'Survived':'nominal', 'Pclass':'ordinal', 'Sex':'nominal',

'Age':'ordinal', 'SibSp':'ordinal', 'Parch':'ordinal',

'Fare':'ordinal', 'Embarked':'nominal'}

def tree\_train(data\_type\_dict, train\_data,test\_data, response, no\_folds,

min\_node\_size, max\_depth, no\_iter):

parameters={'min\_node\_size':min\_node\_size, 'max\_node\_depth':max\_depth,

'threshold':0, 'metric\_kind':'Gini', 'alpha':0,

'response':response}

model=ticart.ClassificationTree()

predictions=[]

for i in range(no\_iter):

for fold in cross\_validate(no\_folds, train\_data):

model=ticart.ClassificationTree()

model.train(fold['train'], data\_type\_dict, parameters, prune=False)

model.load\_new\_data(fold['test'])

model.prune\_tree(alpha=0, new\_data=True)

predictions.append(test\_data.apply(model.predict, axis=1))

return predictions

def combine\_predictions(predictions):

data\_dict ={i:predictions[i] for i in range(len(predictions))}

d=pd.DataFrame(data\_dict)

def mode(x):

key,value = max(x.value\_counts().iteritems(), key=lambda x:x[1])

return key

pred=d.apply(mode, axis=1)

return pred

predictions=tree\_train(data\_type\_dict=data\_type\_dict, train\_data=df,

test\_data=df2, response='Survived', no\_folds=2,

max\_depth=50, min\_node\_size=5, no\_iter=2)

predictions = combine\_predictions(predictions)

prediction\_path = 'C:/Documents and Settings/DIGIT/My Documents/Google Drive/Blogs/triangleinequality/Titanic/prediction.csv'

prediction\_csv=pd.read\_csv(prediction\_path)

prediction\_csv['Survived']=predictions

prediction\_csv.to\_csv('C:/Documents and Settings/DIGIT/My Documents/Google Drive/Blogs/triangleinequality/Titanic/my\_first\_submission2.csv',

index =False)

# specifies the parameters of our graphs

fig = plt.figure(figsize=(18,6), dpi=1600)

alpha=alpha\_scatterplot = 0.2

alpha\_bar\_chart = 0.55

# lets us plot many diffrent shaped graphs together

ax1 = plt.subplot2grid((2,3),(0,0))

# plots a bar graph of those who surived vs those who did not.

df.Survived.value\_counts().plot(kind='bar', alpha=alpha\_bar\_chart)

# this nicely sets the margins in matplotlib to deal with a recent bug 1.3.1

ax1.set\_xlim(-1, 2)

# puts a title on our graph

plt.title("Distribution of Survival, (1 = Survived)")

plt.subplot2grid((2,3),(0,1))

plt.scatter(df.Survived, df.Age, alpha=alpha\_scatterplot)

# sets the y axis lable

plt.ylabel("Age")

# formats the grid line style of our graphs

plt.grid(b=True, which='major', axis='y')

plt.title("Survival by Age, (1 = Survived)")

ax3 = plt.subplot2grid((2,3),(0,2))

df.Pclass.value\_counts().plot(kind="barh", alpha=alpha\_bar\_chart)

ax3.set\_ylim(-1, len(df.Pclass.value\_counts()))

plt.title("Class Distribution")

plt.subplot2grid((2,3),(1,0), colspan=2)

# plots a kernel density estimate of the subset of the 1st class passangers's age

df.Age[df.Pclass == 1].plot(kind='kde')

df.Age[df.Pclass == 2].plot(kind='kde')

df.Age[df.Pclass == 3].plot(kind='kde')

# plots an axis lable

plt.xlabel("Age")

plt.title("Age Distribution within classes")

# sets our legend for our graph.

plt.legend(('1st Class', '2nd Class','3rd Class'),loc='best')

ax5 = plt.subplot2grid((2,3),(1,2))

df.Embarked.value\_counts().plot(kind='bar', alpha=alpha\_bar\_chart)

ax5.set\_xlim(-1, len(df.Embarked.value\_counts()))

# specifies the parameters of our graphs

plt.title("Passengers per boarding location")

plt.figure(figsize=(6,4))

fig, ax = plt.subplots()

df.Survived.value\_counts().plot(kind='barh', color="blue", alpha=.65)

ax.set\_ylim(-1, len(df.Survived.value\_counts()))

plt.title("Survival Breakdown (1 = Survived, 0 = Died)")

fig = plt.figure(figsize=(18,6))

#create a plot of two subsets, male and female, of the survived variable.

#After we do that we call value\_counts() so it can be easily plotted as a bar graph.

#'barh' is just a horizontal bar graph

df\_male = df.Survived[df.Sex == 'male'].value\_counts().sort\_index()

df\_female = df.Survived[df.Sex == 'female'].value\_counts().sort\_index()

ax1 = fig.add\_subplot(121)

df\_male.plot(kind='barh',label='Male', alpha=0.55)

df\_female.plot(kind='barh', color='#FA2379',label='Female', alpha=0.55)

plt.title("Who Survived? with respect to Gender, (raw value counts) "); plt.legend(loc='best')

ax1.set\_ylim(-1, 2)

#adjust graph to display the proportions of survival by gender

ax2 = fig.add\_subplot(122)

(df\_male/float(df\_male.sum())).plot(kind='barh',label='Male', alpha=0.55)

(df\_female/float(df\_female.sum())).plot(kind='barh', color='#FA2379',label='Female', alpha=0.55)

plt.title("Who Survived proportionally? with respect to Gender"); plt.legend(loc='best')

ax2.set\_ylim(-1, 2)

fig = plt.figure(figsize=(18,4), dpi=1600)

alpha\_level = 0.65

# building on the previous code, here we create an additional subset with in the gender subset

# we created for the survived variable. I know, thats a lot of subsets. After we do that we call

# value\_counts() so it it can be easily plotted as a bar graph. this is repeated for each gender

# class pair.

ax1=fig.add\_subplot(141)

female\_highclass = df.Survived[df.Sex == 'female'][df.Pclass != 3].value\_counts()

female\_highclass.plot(kind='bar', label='female, highclass', color='#FA2479', alpha=alpha\_level)

ax1.set\_xticklabels(["Survived", "Died"], rotation=0)

ax1.set\_xlim(-1, len(female\_highclass))

plt.title("Who Survived? with respect to Gender and Class"); plt.legend(loc='best')

ax2=fig.add\_subplot(142, sharey=ax1)

female\_lowclass = df.Survived[df.Sex == 'female'][df.Pclass == 3].value\_counts()

female\_lowclass.plot(kind='bar', label='female, low class', color='pink', alpha=alpha\_level)

ax2.set\_xticklabels(["Died","Survived"], rotation=0)

ax2.set\_xlim(-1, len(female\_lowclass))

plt.legend(loc='best')

ax3=fig.add\_subplot(143, sharey=ax1)

male\_lowclass = df.Survived[df.Sex == 'male'][df.Pclass == 3].value\_counts()

male\_lowclass.plot(kind='bar', label='male, low class',color='lightblue', alpha=alpha\_level)

ax3.set\_xticklabels(["Died","Survived"], rotation=0)

ax3.set\_xlim(-1, len(male\_lowclass))

plt.legend(loc='best')

ax4=fig.add\_subplot(144, sharey=ax1)

male\_highclass = df.Survived[df.Sex == 'male'][df.Pclass != 3].value\_counts()

male\_highclass.plot(kind='bar', label='male, highclass', alpha=alpha\_level, color='steelblue')

ax4.set\_xticklabels(["Died","Survived"], rotation=0)

ax4.set\_xlim(-1, len(male\_highclass))

plt.legend(loc='best')

fig = plt.figure(figsize=(18,12), dpi=1600)

a = 0.65

# Step 1

ax1 = fig.add\_subplot(341)

df.Survived.value\_counts().plot(kind='bar', color="blue", alpha=a)

ax1.set\_xlim(-1, len(df.Survived.value\_counts()))

plt.title("Step. 1")

# Step 2

ax2 = fig.add\_subplot(345)

df.Survived[df.Sex == 'male'].value\_counts().plot(kind='bar',label='Male')

df.Survived[df.Sex == 'female'].value\_counts().plot(kind='bar', color='#FA2379',label='Female')

ax2.set\_xlim(-1, 2)

plt.title("Step. 2 \nWho Survied? with respect to Gender."); plt.legend(loc='best')

ax3 = fig.add\_subplot(346)

(df.Survived[df.Sex == 'male'].value\_counts()/float(df.Sex[df.Sex == 'male'].size)).plot(kind='bar',label='Male')

(df.Survived[df.Sex == 'female'].value\_counts()/float(df.Sex[df.Sex == 'female'].size)).plot(kind='bar', color='#FA2379',label='Female')

ax3.set\_xlim(-1,2)

plt.title("Who Survied proportionally?"); plt.legend(loc='best')

# Step 3

ax4 = fig.add\_subplot(349)

female\_highclass = df.Survived[df.Sex == 'female'][df.Pclass != 3].value\_counts()

female\_highclass.plot(kind='bar', label='female highclass', color='#FA2479', alpha=a)

ax4.set\_xticklabels(["Survived", "Died"], rotation=0)

ax4.set\_xlim(-1, len(female\_highclass))

plt.title("Who Survived? with respect to Gender and Class"); plt.legend(loc='best')

ax5 = fig.add\_subplot(3,4,10, sharey=ax1)

female\_lowclass = df.Survived[df.Sex == 'female'][df.Pclass == 3].value\_counts()

female\_lowclass.plot(kind='bar', label='female, low class', color='pink', alpha=a)

ax5.set\_xticklabels(["Died","Survived"], rotation=0)

ax5.set\_xlim(-1, len(female\_lowclass))

plt.legend(loc='best')

ax6 = fig.add\_subplot(3,4,11, sharey=ax1)

male\_lowclass = df.Survived[df.Sex == 'male'][df.Pclass == 3].value\_counts()

male\_lowclass.plot(kind='bar', label='male, low class',color='lightblue', alpha=a)

ax6.set\_xticklabels(["Died","Survived"], rotation=0)

ax6.set\_xlim(-1, len(male\_lowclass))

plt.legend(loc='best')

ax7 = fig.add\_subplot(3,4,12, sharey=ax1)

male\_highclass = df.Survived[df.Sex == 'male'][df.Pclass != 3].value\_counts()

male\_highclass.plot(kind='bar', label='male highclass', alpha=a, color='steelblue')

ax7.set\_xticklabels(["Died","Survived"], rotation=0)

ax7.set\_xlim(-1, len(male\_highclass))

plt.legend(loc='best')

--- Log regression

# model formula

# here the ~ sign is an = sign, and the features of our dataset

# are written as a formula to predict survived. The C() lets our

# regression know that those variables are categorical.

# Ref: http://patsy.readthedocs.org/en/latest/formulas.html

formula = 'Survived ~ C(Pclass) + C(Sex) + Age + SibSp + C(Embarked)'

# create a results dictionary to hold our regression results for easy analysis later

results = {}

# 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()

# Plot Predictions Vs Actual

plt.figure(figsize=(18,4));

plt.subplot(121, axisbg="#DBDBDB")

# generate predictions from our fitted model

ypred = res.predict(x)

plt.plot(x.index, ypred, 'bo', x.index, y, 'mo', alpha=.25);

plt.grid(color='white', linestyle='dashed')

plt.title('Logit predictions, Blue: \nFitted/predicted values: Red');

# Residuals

ax2 = plt.subplot(122, axisbg="#DBDBDB")

plt.plot(res.resid\_dev, 'r-')

plt.grid(color='white', linestyle='dashed')

ax2.set\_xlim(-1, len(res.resid\_dev))

plt.title('Logit Residuals');

fig = plt.figure(figsize=(18,9), dpi=1600)

a = .2

# Below are examples of more advanced plotting.

# It it looks strange check out the tutorial above.

fig.add\_subplot(221, axisbg="#DBDBDB")

kde\_res = KDEUnivariate(res.predict())

kde\_res.fit()

plt.plot(kde\_res.support,kde\_res.density)

plt.fill\_between(kde\_res.support,kde\_res.density, alpha=a)

plt.title("Distribution of our Predictions")

fig.add\_subplot(222, axisbg="#DBDBDB")

plt.scatter(res.predict(),x['C(Sex)[T.male]'] , alpha=a)

plt.grid(b=True, which='major', axis='x')

plt.xlabel("Predicted chance of survival")

plt.ylabel("Gender Bool")

plt.title("The Change of Survival Probability by Gender (1 = Male)")

fig.add\_subplot(223, axisbg="#DBDBDB")

plt.scatter(res.predict(),x['C(Pclass)[T.3]'] , alpha=a)

plt.xlabel("Predicted chance of survival")

plt.ylabel("Class Bool")

plt.grid(b=True, which='major', axis='x')

plt.title("The Change of Survival Probability by Lower Class (1 = 3rd Class)")

fig.add\_subplot(224, axisbg="#DBDBDB")

plt.scatter(res.predict(),x.Age , alpha=a)

plt.grid(True, linewidth=0.15)

plt.title("The Change of Survival Probability by Age")

plt.xlabel("Predicted chance of survival")

plt.ylabel("Age")

-

test\_data['Survived'] = 1.23

results

# Use your model to make prediction on our test set.

compared\_resuts = ka.predict(test\_data, results, 'Logit')

compared\_resuts = Series(compared\_resuts) # convert our model to a series for easy output

# output and submit to kaggle

compared\_resuts.to\_csv("data/output/logitregres.csv")

# Create an acceptable formula for our machine learning algorithms

formula\_ml = 'Survived ~ C(Pclass) + C(Sex) + Age + SibSp + Parch + C(Embarked)'

# set plotting parameters

plt.figure(figsize=(8,6))

# create a regression friendly data frame

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

# select which features we would like to analyze

# try chaning the selection here for diffrent output.

# Choose : [2,3] - pretty sweet DBs [3,1] --standard DBs [7,3] -very cool DBs,

# [3,6] -- very long complex dbs, could take over an hour to calculate!

feature\_1 = 2

feature\_2 = 3

X = np.asarray(x)

X = X[:,[feature\_1, feature\_2]]

y = np.asarray(y)

# needs to be 1 dimenstional so we flatten. it comes out of dmatirces with a shape.

y = y.flatten()

n\_sample = len(X)

np.random.seed(0)

order = np.random.permutation(n\_sample)

X = X[order]

y = y[order].astype(np.float)

# do a cross validation

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:]

# create a list of the types of kerneks we will use for your analysis

types\_of\_kernels = ['linear', 'rbf', 'poly']

# specify our color map for plotting the results

color\_map = plt.cm.RdBu\_r

# fit the model

for fig\_num, kernel in enumerate(types\_of\_kernels):

clf = svm.SVC(kernel=kernel, gamma=3)

clf.fit(X\_train, y\_train)

plt.figure(fig\_num)

plt.scatter(X[:, 0], X[:, 1], c=y, zorder=10, cmap=color\_map)

# circle out the test data

plt.scatter(X\_test[:, 0], X\_test[:, 1], s=80, facecolors='none', zorder=10)

plt.axis('tight')

x\_min = X[:, 0].min()

x\_max = X[:, 0].max()

y\_min = X[:, 1].min()

y\_max = X[:, 1].max()

XX, YY = np.mgrid[x\_min:x\_max:200j, y\_min:y\_max:200j]

Z = clf.decision\_function(np.c\_[XX.ravel(), YY.ravel()])

# put the result into a color plot

Z = Z.reshape(XX.shape)

plt.pcolormesh(XX, YY, Z > 0, cmap=color\_map)

plt.contour(XX, YY, Z, colors=['k', 'k', 'k'], linestyles=['--', '-', '--'],

levels=[-.5, 0, .5])

plt.title(kernel)

plt.show()

# Here you can output which ever result you would like by changing the Kernel and clf.predict lines

# Change kernel here to poly, rbf or linear

# adjusting the gamma level also changes the degree to which the model is fitted

clf = svm.SVC(kernel='poly', gamma=3).fit(X\_train, y\_train)

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

# Change the interger values within x.ix[:,[6,3]].dropna() explore the relationships between other

# features. the ints are column postions. ie. [6,3] 6th column and the third column are evaluated.

res\_svm = clf.predict(x.ix[:,[6,3]].dropna())

res\_svm = DataFrame(res\_svm,columns=['Survived'])

res\_svm.to\_csv("data/output/svm\_poly\_63\_g10.csv") # saves the results for you, change the name as you please.

# 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)