Individual

by Chang-Jung Wu

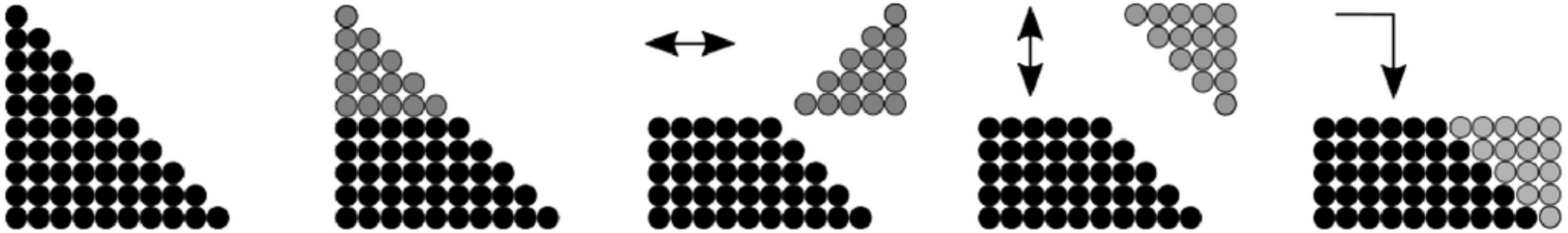
FILE	INDIVIDUAL_2687899_1492965966.PDF (629.93K)		
TIME SUBMITTED	11-DEC-2016 10:59AM	WORD COUNT	6703
SUBMISSION ID	63833469	CHARACTER COUNT	34284

MSING055 Individual Coursework: Gauss, Palindromes & The Titanic

Solve these first two exercises to show your familiarity with functions, decisions and string manipulation in Python

Little Gauss problem

You may have heard that when Carl Friedrich Gauss (https://en.wikipedia.org/wiki/Carl_Friedrich_Gauss) (1777-1855) was in primary school, one day his teacher wanted to keep busy the class with a very long and tedious task. The teacher asked to add together all the number from 1 to 100, but little Gauss stood up and gave the answer just after few seconds: *"5050"* Gauss said. The teacher couldn't believe it, but Gauss told him it was a very easy problem. By adding the numbers in pairs, the last with the first and so on (*1 + 100, 2 + 99, ...*) he found that the result where always the same (*101, 101, ...*) so he has only to multiply that by 50.



Though that's a very smart and quick way to solve the problem, try to solve it by brute force - *i.e.* as Gauss's teacher had planned the kids sould have done it, but with an extra twist, just add the even numbers.

1. Write a function called `gauss_friend_loop` that uses a plain `for` loop and it can be used for any range of positive numbers (including the number you input), the default value should be `100` if not input is set. Write comments through the code to help us understand your reasoning.

In [1]:
```
def gauss_friend_loop(default = 100):
    # Initizalising a new variable for the output answer
    answer = 0

    # Looping the given number, but it shall be noticed that range 1 to n is range(1,n+1)
    for n in range(1, default + 1):
        # Checking whether the number is odd
        if n % 2 != 0:
            answer += n # Add to our counter
    return answer
```

2. Write a function called `gauss_friend_comprehension` that do the same than `gauss_friend_loop` but using a list comprehension instead of a classic `for` loop.

In [2]:
```
def gauss_friend_comprehension(default = 100):
    # List comprehension
    answer = [a for a in range(1,default+1) if a%2!=0]
    # Summing the answer
    return sum(answer)
```

Either of these two functions has to pass the following tests:

In [3]:
```
assert gauss_friend_loop() == gauss_friend_comprehension()
assert gauss_friend_loop(99) == gauss_friend_loop()
assert gauss_friend_comprehension(3) == 4
```

Palindromes

Some times you find beautiful pieces of texts that can be read in both directions and it says the same, they are called palindromes (https://en.wikipedia.org/wiki/Palindrome). Some are simple words such as *mum*, *dad* or *kayak*, whereas others are more sophisticated sentences: *Never odd or even*, *A nut for a jar of tuna*, etc.

1. Write a function called `is_palindrome` that returns a boolean if the input is a palindrome. Just consider letter and numbers, forget about punctuation marks (*i.e.*, ', ?, !, ,, ...), though they may be input by the user.

In [4]:
```
def is_palindrome(text):
    # Only leaving characters and making them lowercase
    newText = ''.join(p for p in str(text) if p.isalnum()).lower()
    # Reversing the newText
    reverse = ''.join(p for p in str(text) if p.isalnum()).lower()[::-1]
    if reverse == newText:
        return True
    else:
        return False
```

In [5]:
```
assert is_palindrome('Anna')
assert not is_palindrome('David')
assert is_palindrome('A man, a plan, a canal: Panama.')
assert is_palindrome('343')
assert is_palindrome(343)
```

You can create more tests from this huge palindrome list (http://www.palindromelist.net/).

Titanic Machine Learning from Disaster

From the Kaggle competition homepage (http://www.kaggle.com/c/titanic-gettingStarted):

> The sinking of the RMS Titanic is one of the most infamous shipwrecks in history. On April 15, 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.
>
> In this contest, we ask you to complete the analysis of what sorts of people were likely to survive. In particular, we ask you to apply the tools of machine learning to predict which passengers survived the tragedy.
>
> This Kaggle Getting Started Competition provides an ideal starting place for people who may not have a lot of experience in data science and machine learning."

Show proficiecy with Python, Jupyter notebooks and SciPy/PyData libraries (NumPy, Pandas, SciKit-Learn) undertaking a simple example of an analysis of the Titanic disaster dataset from Kaggle.

This third exercise of the coursework will cover basic examples of:

Data Preparation

- Importing Data with Pandas
- Cleaning Data

Exploratory Data Analysis

- Exploring Data through Visualizations with Matplotlib

Supervised Machine learning

- Training a Basic Random Forest

Required Libraries:

- NumPy (http://www.numpy.org/)
- IPython (http://ipython.org/)
- Pandas (http://pandas.pydata.org/)

- SciKit-Learn (http://scikit-learn.org/stable/)
- SciPy (http://www.scipy.org/)
- Matplotlib (http://matplotlib.org/)

Data Wrangling

Importing Data with Pandas

In [6]:
```
# Magic to show the plots within the notebook:
%matplotlib inline

# Import libraries to be used
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
```

In [7]:
```
# Read the csv file (see Lecture notes for tips)
df = pd.read_csv('data/train.csv')

# For another analysis
from copy import deepcopy
titanicDF = deepcopy(df) # for plotting
svmDF = deepcopy(df) # for SVM and Missing values
```

Show an overview of our data:

In [8]:
```
# Now inspect your dataframe
df
```

Out[8]:

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
0	1	0	3	Braund, Mr. Owen Harris	male	22.0	1	0	A/5 21171	7.2500	NaN	S
1	2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	0	PC 17599	71.2833	C85	C
2	3	1	3	Heikkinen, Miss. Laina	female	26.0	0	0	STON/O2. 3101282	7.9250	NaN	S
3	4	1	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0	1	0	113803	53.1000	C123	S
4	5	0	3	Allen, Mr. William Henry	male	35.0	0	0	373450	8.0500	NaN	S
5	6	0	3	Moran, Mr. James	male	NaN	0	0	330877	8.4583	NaN	Q
6	7	0	1	McCarthy, Mr. Timothy J	male	54.0	0	0	17463	51.8625	E46	S
				Palsson, Master. Gosta								

Above is a summary of our data contained in a Pandas dataframe.

We can get more information about the dataframe using `.info()`

In [9]:
```
# Get more information about the dataframe
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 891 entries, 0 to 890
Data columns (total 12 columns):
PassengerId    891 non-null int64
Survived       891 non-null int64
Pclass         891 non-null int64
Name           891 non-null object
Sex            891 non-null object
Age            714 non-null float64
SibSp          891 non-null int64
Parch          891 non-null int64
Ticket         891 non-null object
Fare           891 non-null float64
Cabin          204 non-null object
Embarked       889 non-null object
dtypes: float64(2), int64(5), object(5)
memory usage: 83.6+ KB
```

As you can see the summary holds quite a bit of information. First, it lets us know we have 891 observations, or passengers, to analyze here:

```
Int64Index: 891 entries, 0 to 890
```

Next it shows us all of the columns in `DataFrame`. Each column tells us something about each of our observations, like their `name`, `sex` or `age`. These colunms are called a features of our dataset. You can think of the meaning of the words column and feature as interchangeable for this notebook.

After each feature it lets us know how many values it contains. While most of our features have complete data on every observation, like the `survived` feature here:

```
survived    891  non-null values
```

some are missing information, like the `age` feature:

```
age         714  non-null values
```

These missing values are represented as `NaN`s.

We can describe the feature in our dataframe and get some basic statistics using `.describe()`

In [10]:
```
# Get descriptive statistics about the dataframe
df.describe()
```

```
C:\Users\user\Anaconda3\lib\site-packages\numpy\lib\function_base.py:3834: RuntimeWarning: Invalid value encountered in
 percentile
  RuntimeWarning)
```

Out[10]:

	PassengerId	Survived	Pclass	Age	SibSp	Parch	Fare
count	891.000000	891.000000	891.000000	714.000000	891.000000	891.000000	891.000000
mean	446.000000	0.383838	2.308642	29.699118	0.523008	0.381594	32.204208
std	257.353842	0.486592	0.836071	14.526497	1.102743	0.806057	49.693429
min	1.000000	0.000000	1.000000	0.420000	0.000000	0.000000	0.000000
25%	223.500000	0.000000	2.000000	NaN	0.000000	0.000000	7.910400
50%	446.000000	0.000000	3.000000	NaN	0.000000	0.000000	14.454200
75%	668.500000	1.000000	3.000000	NaN	1.000000	0.000000	31.000000
max	891.000000	1.000000	3.000000	80.000000	8.000000	6.000000	512.329200

Cleaning Data

The features `ticket` and `cabin` have many missing values and so can't add much value to our analysis. To handle this we will drop them from the dataframe to preserve the integrity of our dataset.

To do that we'll use this line of code to drop the features entirely:

```
df = df.drop(['feature_1','feature_2'], axis=1)
```

In [11]:
```
```

```
# Drop Ticket and Cabin columns
df = df.drop(['Ticket', 'Cabin'], axis=1)
#axis=1 denotes that we are referring to a column, not a row
    df = df.dropna()
```

While this line of code removes the NaN values from every remaining feature using `.dropna()`:

In [12]:
```
# Remove NaN values
df = df.dropna()
```

Now we have a clean and tidy dataset that is ready for analysis. Because `.dropna()` removes an observation from our data even if it only has 1 `NaN` in one of the features, it would have removed most of our dataset if we had not dropped the `ticket` and `cabin` features first.

Exploratory Data Analysis

Exploring Data through Visualizations with Matplotlib

We are interested in predicting if an individual will survive based on the features in the data like:

- Passenger class ("Pclass" in the data)
- Sex
- Age
- Fare price

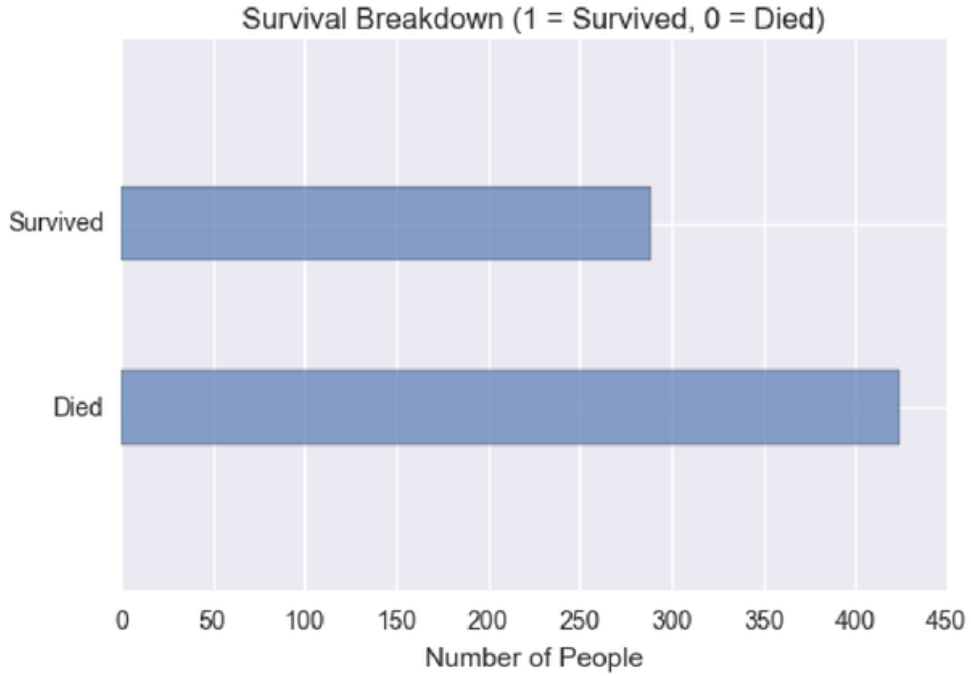
Let's see if we can gain a better understanding of who survived and died.

First let's plot a bar graph of those who survived versus those who were less fortunate.

In [13]:
```
# Importing another package
import seaborn as sns
```

In [14]:
```
# Selecting dataframes by their features---Survived == 1
saved = df[df['Survived'] == 1]

# Selecting dataframes by their features---Survived == 0
dead = df[df['Survived'] == 0]

# Initialising the matplotlib figure
f, ax = plt.subplots(1, 1, figsize=(6, 4))

# Drawing the barplot
plt.barh([0,1], [dead.shape[0], saved.shape[0]],align='center', \
         alpha=0.65, height = 0.4, tick_label = ['Died', 'Survived'])
#matplotlib.pyplot.barh(bottom, width, height=0.8, left=None, hold=None, **kwargs)
plt.xlabel('Number of People')
ax.set_ylim(-1, 2)
plt.title('Survival Breakdown (1 = Survived, 0 = Died)')
```

Out[14]:
```
<matplotlib.text.Text at 0x13469e38390>
```

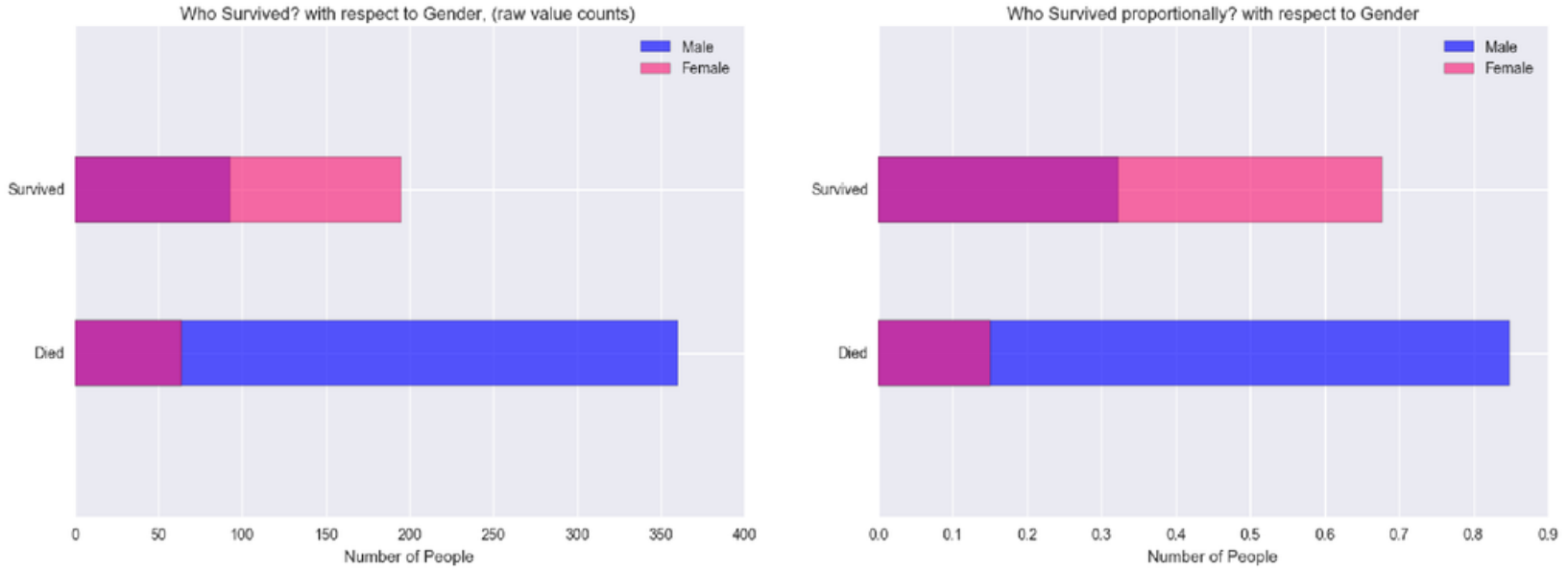


Now let's tease more structure out of the data, let's break the previous graph down by gender

In [15]:

```
# Initialising the matplotlib figure
fig = plt.figure(figsize=(18,6))

# Subplot 1
ax1 = fig.add_subplot(121)

# Plotting the Survival barplot based on Male
plt.barh([0,1], [dead[dead['Sex'] == 'male'].shape[0], saved[saved['Sex'] == 'male'].shape[0]], \
         align='center', alpha=0.65, height = 0.4, \
         tick_label = ['Died', 'Survived'], color = 'blue', label = 'Male')

# Plotting the Survival barplot based on Female
plt.barh([0,1], [dead[dead['Sex'] == 'female'].shape[0], saved[saved['Sex'] == 'female'].shape[0]], \
         align='center', alpha=0.65, height = 0.4, \
         tick_label = ['Died', 'Survived'], color = '#FA2379', label = 'Female')

# Label & Title
plt.xlabel('Number of People')
ax1.set_ylim(-1, 2)
ax1.set_title("Who Survived? with respect to Gender, (raw value counts) ")
plt.legend()

# Subplot 2
ax2 = fig.add_subplot(122)

# Survival and Dead Rate based on Sex
deadMenRate = dead[dead['Sex'] == 'male'].shape[0] / dead.shape[0]
deadWomenRate = dead[dead['Sex'] == 'female'].shape[0] / dead.shape[0]
savedMenRate = saved[saved['Sex'] == 'male'].shape[0] / saved.shape[0]
savedWomenRate = saved[saved['Sex'] == 'female'].shape[0] / saved.shape[0]

# Plotting the Survival barplot based on Male
plt.barh([0,1], [deadMenRate, savedMenRate], \
         align='center', alpha=0.65, height = 0.4, \
         tick_label = ['Died', 'Survived'], color = 'blue', label = 'Male')

# Plotting the Survival barplot based on Female
plt.barh([0,1], [deadWomenRate, savedWomenRate], \
         align='center', alpha=0.65, height = 0.4, \
         tick_label = ['Died', 'Survived'], color = '#FA2379', label = 'Female')

# Label & Title
```

```
plt.xlabel('Rate')
ax2.set_ylim(-1, 2)
ax2.set_title("Who Survived proportionally? with respect to Gender")

plt.legend()
plt.show()
```



Here it's clear that although more men died and survived in raw value counts, females had a greater survival rate proportionally (~75%), than men (~20%)

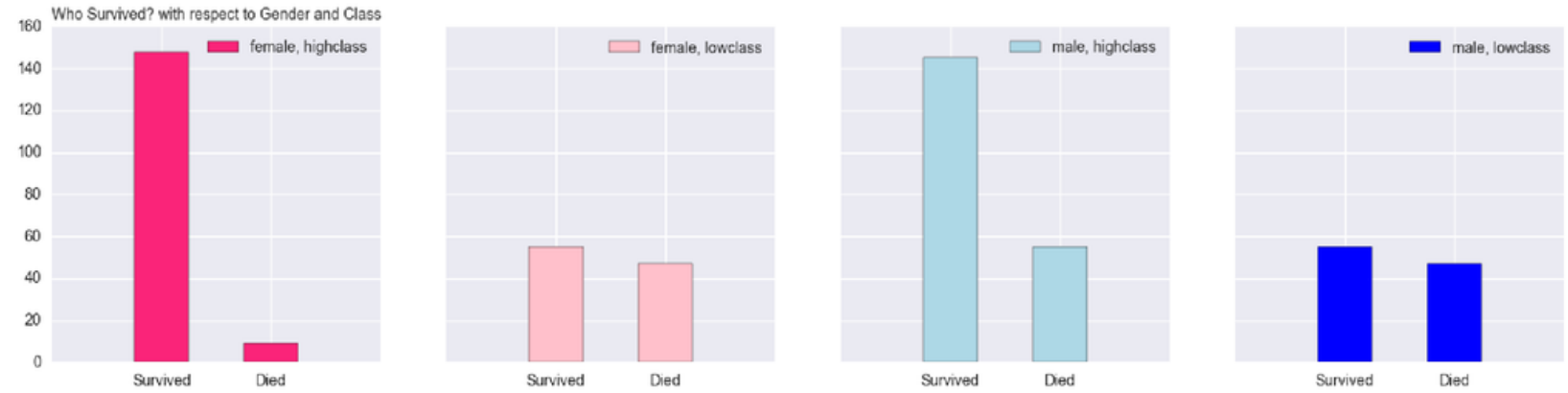
Can we capture more of the structure by using `Pclass`? Here we will bucket classes as lowest class or any of the high classes (classes `1` and `2`). `3` is lowest class. Let's break it down by Gender and what Class they were traveling in.

Building on the previous plot, here we create an additional subset with in the gender subset we created for the survived variable. Then we call `value_counts()` so it it can be easily plotted as a bar graph. This is repeated for each gender class pair.

In [16]:

```
## Making new dataframes by classifying
maleHighClass = df.Survived[df.Sex == 'male'][df.Pclass != 3].value_counts()
maleLowClass = df.Survived[df.Sex == 'female'][df.Pclass == 3].value_counts()
femaleHighClass = df.Survived[df.Sex == 'female'][df.Pclass != 3].value_counts()
femaleLowClass = df.Survived[df.Sex == 'female'][df.Pclass == 3].value_counts()

## Initialising the matplotlib figure
fig = plt.figure(figsize=(18,4))

# Subplot 1
ax1 = fig.add_subplot(141)

# Plotting the Survival barplot --- Female & High Class
femaleHighClass.plot(kind = 'bar',color='#FA2479', label='female, highclass')
ax1.set_xticklabels(["Survived", "Died"], rotation = 0)
ax1.set_xlim(-1, len(femaleHighClass))
ax1.set_title("Who Survived? with respect to Gender and Class", fontdict = {'fontsize': 10})
plt.legend()


## Subplot 2
ax2 = fig.add_subplot(142, sharey=ax1)

# Plotting the Survival barplot --- Female & Low Class
femaleLowClass.plot(kind = 'bar',color='pink', label='female, lowclass')
ax2.set_xticklabels(["Survived", "Died"], rotation = 0)
ax2.set_xlim(-1, len(femaleLowClass))
plt.legend()


## Subplot 3
ax3 = fig.add_subplot(143, sharey=ax1)
# Plotting the Survival barplot --- Male & High Class
maleHighClass.plot(kind = 'bar',color='lightblue', label='male, highclass')
ax3.set_xticklabels(["Survived", "Died"], rotation = 0)
ax3.set_xlim(-1, len(maleHighClass))
plt.legend()


## Subplot 4
ax4 = fig.add_subplot(144, sharey=ax1)
# Plotting the Survival barplot --- Male & Low Class
```

```
maleLowClass.plot(kind = 'bar',color='blue', label='male, lowclass')
ax4.set_xticklabels(["Survived", "Died"], rotation=0)
ax4.set_xlim(-1, len(maleLowClass))
plt.legend()


plt.show()
```



Now we have a lot more information on who survived and died in the tragedy.

With this deeper understanding, we are better equipped to create better more insightful models.

This is a typical process in interactive data analysis. First you start small and understand the most basic relationships and slowly increment the complexity of your analysis as you discover more and more about the data you're working with.

Now that we have a basic understanding of what we are trying to predict, let's predict it.

Supervised Machine Learning

Random Forest

From Wikipedia:

> 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.

Random Forest are a form of non-parametric modeling.

A random forest algorithm randomly generates many simple decision tree models to explain the variance observed in random subsections of our data. These models are may perform poorly individually but once they are averaged, they can be powerful predictive tools. The averaging step is important. 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.

Below we show the process of instantiating and fitting a random forest, scoring the results and generating feature importances.

Training a Basic Random Forest

Firstly we need to import the `RandonForestClassifier` and `preprocessing` modules from the Scikit-learn library

In [17]:
```
from sklearn.ensemble import RandomForestClassifier
from sklearn import preprocessing
```

We need to seed the random seed and initialize the label encoder in order to preprocess some of our features

In [18]:
```
# Set the random seed
np.random.seed(12)

# Initialize label encoder
label_encoder = preprocessing.LabelEncoder()
```

Next, we will convert the categorical features `Sex` and `Embarked` to numerical values using the label encoder `fit_transform` function as follows:

```
df["feature"] = label_encoder.fit_transform(df["feature"])
```

In [19]:
```
# Convert the Sex and Embarked features to numeric values
df["Sex"] = label_encoder.fit_transform(df["Sex"])
df["Embarked"] = label_encoder.fit_transform(df["Embarked"])
```

We need to initialize the Random Forest model with the following parameters: number of estimators as `1000`, max features as `2` and oob_score as `True`, see the Sci-kit learn documentation for RandomForestClassifier (http://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html) and User Guide (http://scikit-learn.org/stable/modules/ensemble.html#forests-of-randomized-trees) for usage.

In [20]:
```
# Initialize the Random Forest model
rfModel = RandomForestClassifier(n_estimators=1000, max_features= 2, oob_score = True)

#sklearn.ensemble.RandomForestClassifier(n_estimators=10, criterion='gini', \
#    max_depth=None, min_samples_split=2, min_samples_leaf=1, \
#    min_weight_fraction_leaf=0.0, max_features='auto', max_leaf_nodes=None, \
#    min_impurity_split=1e-07, bootstrap=True, oob_score=False, n_jobs=1, \
#    random_state=None, verbose=0, warm_start=False, class_weight=None)[source]
```

Define our features to be used and train the Random Forest model using the `.fit()` function and passing `X=df[features]` and `y=df["Survived"])` as arguments.

In [21]:
```
# Define our features
features = ["Sex", "Pclass", "SibSp", "Embarked", "Age", "Fare"]

# Train the model
rfModel.fit(X = df[features], y = df['Survived'])
```

Out[21]:
```
RandomForestClassifier(bootstrap=True, class_weight=None, criterion='gini',
            max_depth=None, max_features=2, max_leaf_nodes=None,
            min_impurity_split=1e-07, min_samples_leaf=1,
            min_samples_split=2, min_weight_fraction_leaf=0.0,
            n_estimators=1000, n_jobs=1, oob_score=True, random_state=None,
            verbose=0, warm_start=False)
```

Finally we print the out-of-bag accuracy using `rf_model.oob_score_` followed by the feature importances.

In [22]:
```
# Print OOB accuracy
print("OOB accuracy: %s" % (rfModel.oob_score_) )
```

```
OOB accuracy: 0.796348314607
```

In [23]:
```
# Print feature importances
for feature, importance in zip(features, rfModel.feature_importances_):
    print("%s: %s" % (feature, importance))
```

```
Sex: 0.256105121864
Pclass: 0.0984827750018
SibSp: 0.0478944245556
Embarked: 0.0266656647565
Age: 0.300806975508
Fare: 0.270045038315
```

Type *Markdown* and LaTeX: α^2

Additional

Outline:

- Random Forests
- Plotting
- Missing Values
- SVM & Cross Validation

1. Random Forests--- Continue

Even though we found out the importance value of each feature, we can sort these features by their values to clearly see which features are more important than the others.

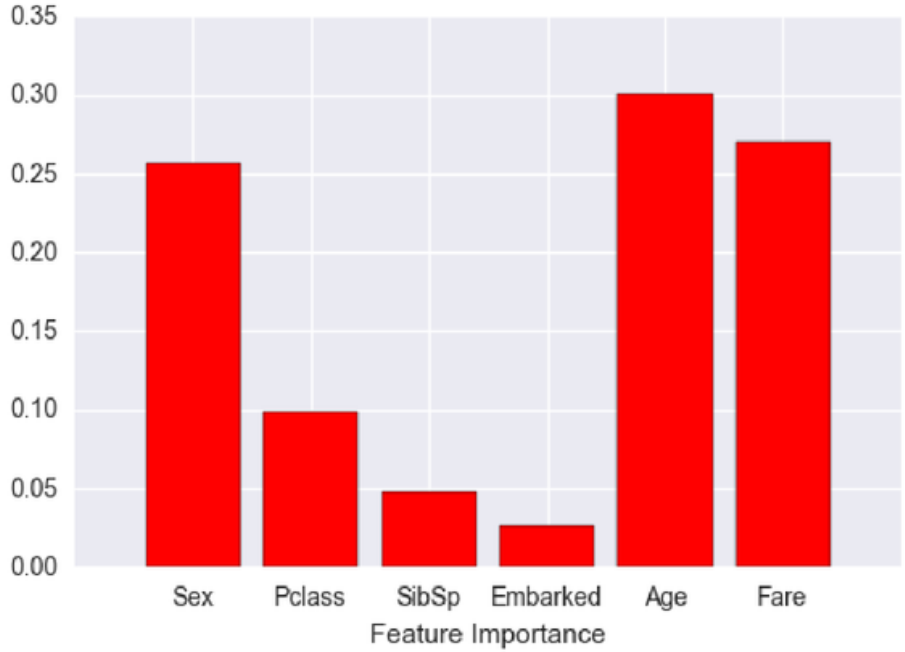
In [24]:
```
# Print feature importances (Already done in previous section)
importanceDictionary = {}
for feature, importance in zip(features, rfModel.feature_importances_):
    importanceDictionary[str(feature)] = importance

# Sort the features by the most important to the least
for w in sorted(importanceDictionary, key=importanceDictionary.get, reverse=True):
    print(w, importanceDictionary[w])
```

```
Age 0.300806975508
Fare 0.270045038315
Sex 0.256105121864
Pclass 0.0984827750018
SibSp 0.0478944245556
Embarked 0.0266656647565
```

Also, we can visualise these features and their values by plotting them. In order to plot them, we need to make those key-value pairs become arrary.

In [25]:
```
'''
## Reminder
features = ["Sex", "Pclass", "SibSp", "Embarked", "Age", "Fare"]
importanceDictionary = {}
for feature, importance in zip(features, rfModel.feature_importances_):
    importanceDictionary[str(feature)] = importance
'''
importances = rfModel.feature_importances_
np.ndarray.tolist(importances);
```

In [26]:
```
# Initialising the matplotlib figure
f, ax = plt.subplots(1, 1, figsize=(6, 4))

plt.bar(range(0, len(features)), importances, \
       color="r", align="center")

plt.xlabel('Feature Importance')
ax.set_xticklabels(["Sex", "Pclass", "SibSp", "Embarked", "Age", "Fare"])
ax.set_xticks(range(len(features)+1))
plt.show()
```



As we can see, the top three important features are **Age**, **Fare**, **Sex**.

Type *Markdown* and LaTeX: α^2

2. Plotting

From a sample of the Titanic data, we can see the various features present for each passenger on the ship:

- **Survived**: Outcome of survival (0 = No; 1 = Yes)
- **Pclass**: Socio-economic class (1 = Upper class; 2 = Middle class; 3 = Lower class)
- **Name**: Name of passenger
- **Sex**: Sex of the passenger
- **Age**: Age of the passenger (Some entries contain NaN)
- **SibSp**: Number of siblings and spouses of the passenger aboard
- **Parch**: Number of parents and children of the passenger aboard
- **Ticket**: Ticket number of the passenger
- **Fare**: Fare paid by the passenger
- **Cabin**: Cabin number of the passenger (Some entries contain NaN)
- **Embarked**: Port of embarkation of the passenger (C = Cherbourg; Q = Queenstown; S = Southampton)

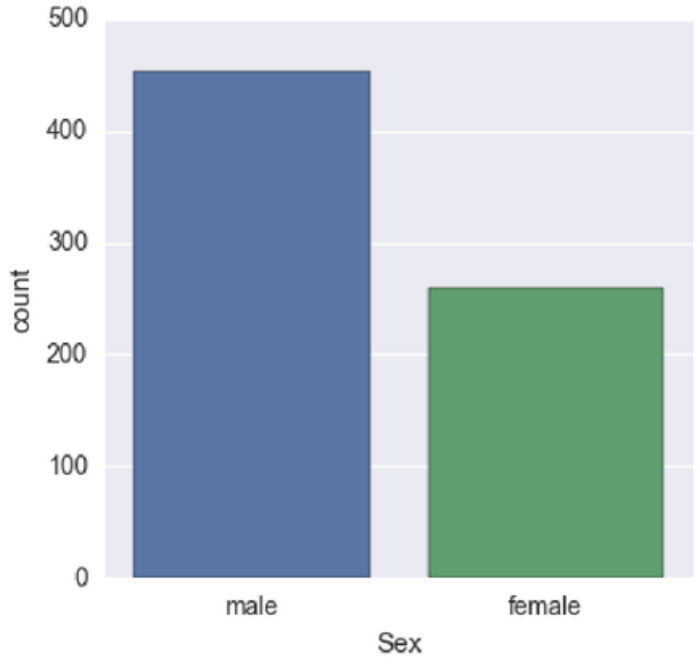
A. Descriptive Plots

Since we have been working with the *df* dataframe, we need another dataframe that has not been processed by us. Previously, we copied the df dataframe and named it ***titanicDF***. In this plotting section, we will use this ***titanicDF*** instead.

In [27]:
```
# Drop Ticket and Cabin columns and remove Nan values
titanicDF = titanicDF.drop(['Ticket','Cabin'], axis=1).dropna()
```

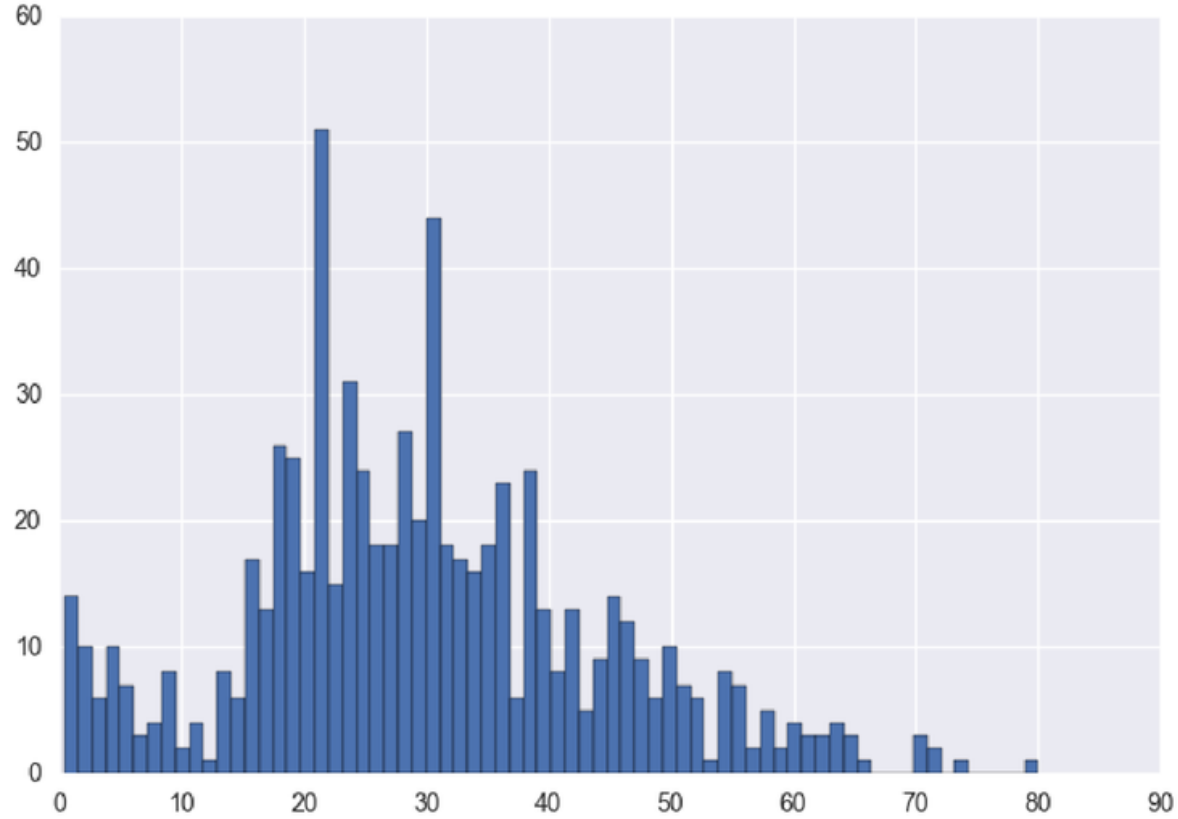
In [28]:
```
# Let's first look at gender
sns.factorplot('Sex', data=titanicDF, kind = 'count')
```

Out[28]: <seaborn.axisgrid.FacetGrid at 0x1346a748a20>



In [29]:
```
# Let's look at the distribution of age
titanicDF['Age'].hist(bins=70)
```

Out[29]: <matplotlib.axes._subplots.AxesSubplot at 0x1346ac737f0>



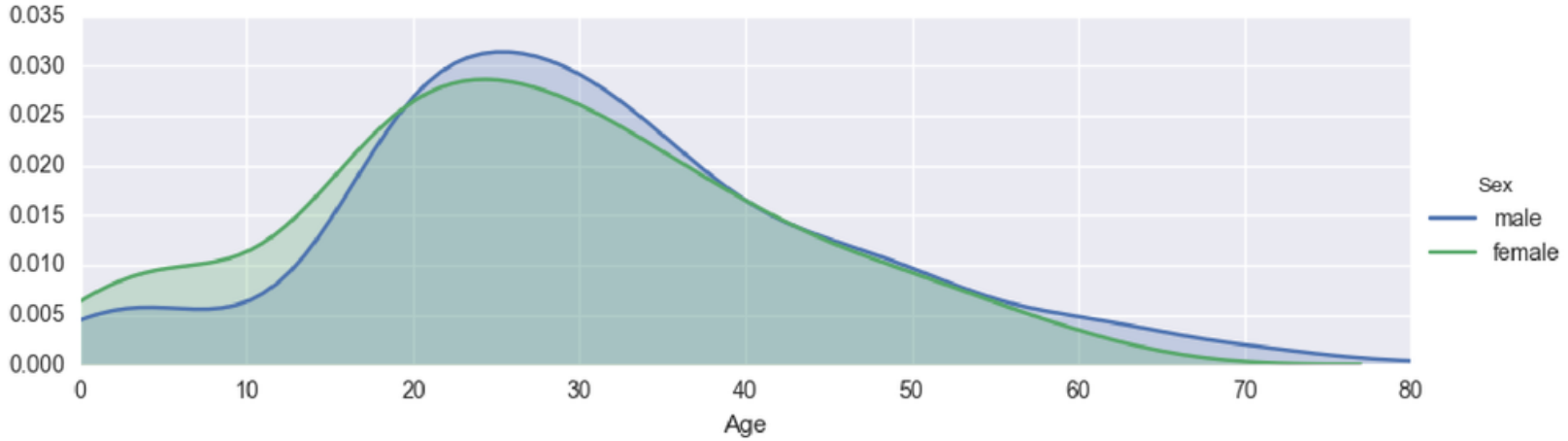
From the figure we can see that most of the passangers were aged between 20 to 30 year old. If we want to see more deeper about what were the distributions looked like if we divide them into groups of **Pclass** and **Sex**.

Age distribution grouped by sex

In [30]:
```
# Another way to visualize the data is to use FacetGrid to plot multiple kedplots on one plot
# Set the figure equal to a facetgrid with the pandas dataframe as its data source,
#set the hue, and change the aspect ratio.
fig = sns.FacetGrid(titanicDF, hue="Sex",aspect=3)

# Next use map to plot all the possible kdeplots for the 'Age' column by the hue choice
fig.map(sns.kdeplot,'Age',shade= True)

# Age starts from 0 to the oldest age in the dataset
fig.set(xlim=(0, titanicDF['Age'].max()))

# Add a legend
fig.add_legend()
```

```
C:\Users\user\Anaconda3\lib\site-packages\statsmodels\nonparametric\kdetools.py:20: VisibleDeprecationWarning: using a
 non-integer number instead of an integer will result in an error in the future
  y = X[:m/2+1] + np.r_[0,X[m/2+1:],0]*1j
```

Out[30]: <seaborn.axisgrid.FacetGrid at 0x1346adf6828>



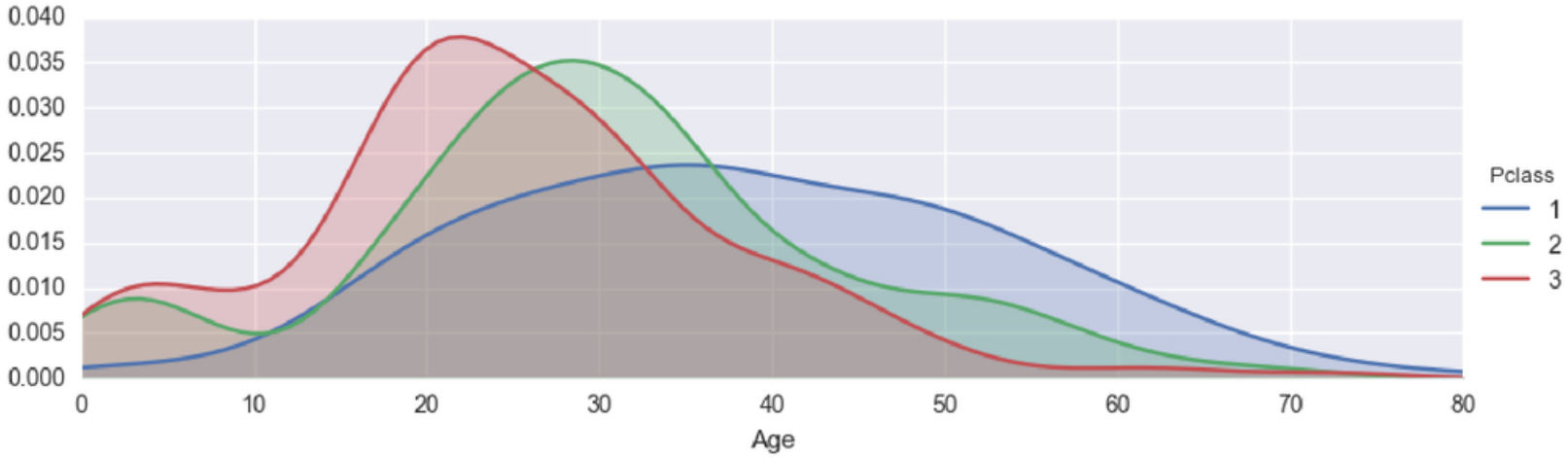
We can see that male and female had similar age distribution, which centralised between age 20 to age 30.

Age distribution grouped by Pclass

In [31]:
```
# Do the similar thing, but this time we need to change hue="Sex"
fig = sns.FacetGrid(titanicDF, hue="Pclass", aspect=3)
fig.map(sns.kdeplot,'Age',shade= True)
oldest = titanicDF['Age'].max()
fig.set(xlim=(0,titanicDF['Age'].max()))
fig.add_legend()
```

```
C:\Users\user\Anaconda3\lib\site-packages\statsmodels\nonparametric\kdetools.py:20: VisibleDeprecationWarning: using a
 non-integer number instead of an integer will result in an error in the future
  y = X[:m/2+1] + np.r_[0,X[m/2+1:],0]*1j
```

Out[31]: <seaborn.axisgrid.FacetGrid at 0x1346ae9deb8>

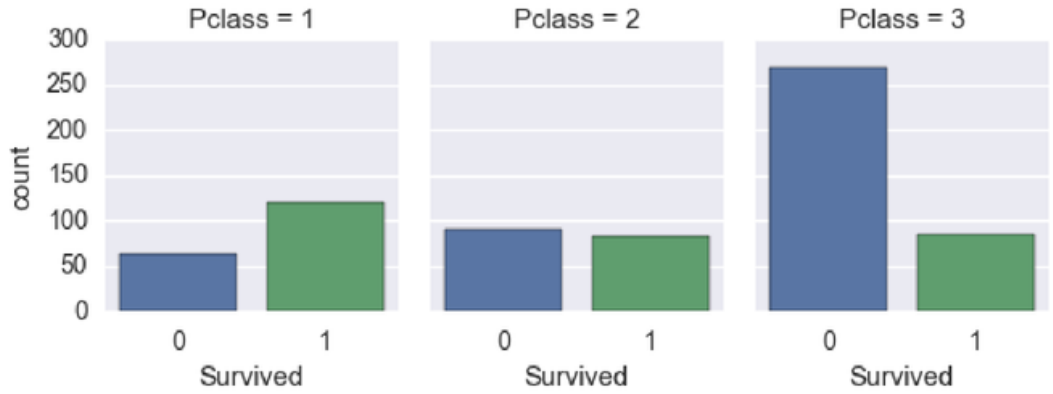


Comparing to the previous figure, in this figure we can see that the distributions between each Pclass had slightly different to each others. People in Pclass 3 were younger than the other two on average. While more people from Plcass 2 and Pclass 3 aged between 20 to 30 year old, there were more people aged 30 to 40 year old with Pclass1; also, the age of Plcass 1 were more evenly distributed comparing to other two groups.

B. Pclass VS. Survived

In [32]:
```
'''
## Reminder: We already import seaborn as sns in the previous section
'''
# Making Barplot of Pclass VS. Survived
sns.factorplot("Survived", col="Pclass",  data=df, kind="count", size=2.5, aspect=.8)
```

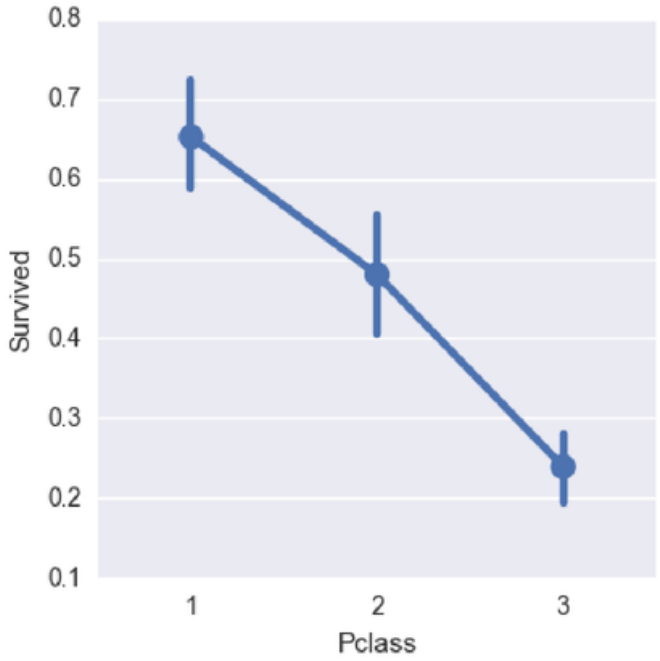
Out[32]: <seaborn.axisgrid.FacetGrid at 0x1346af262e8>



Through this visuaisation, we can see that more people survived in the Pclass 1 than the people in the Pclass 3. However, we would like to see the survival rate of each Pclass, so that we can compare the rate more easily.

In [33]:
```
# Comparing Survival rate between Pclass and Survived
sns.factorplot('Pclass','Survived',data=titanicDF)
```

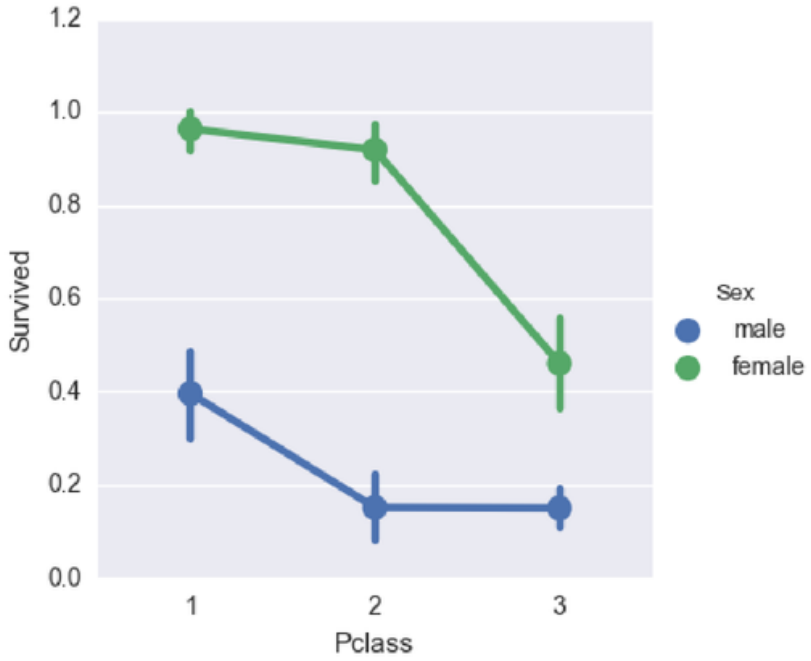
Out[33]: <seaborn.axisgrid.FacetGrid at 0x1346d2a6198>



We can clearly see the different survival rate in each Pclass. People in Pclass 1 had the highest survival rate, and then Pclass 2 the second, Pclass 3 the last. As we implemented ***Random Forest***, we found that **'Sex'** was the third important feature. Therefore, let's take a look at the survival rate for both gender in different Pclass.

In [34]:
```
# Survival Rate for both gender in different Pclass
sns.factorplot('Pclass','Survived', hue = 'Sex',data=titanicDF)
```

Out[34]: <seaborn.axisgrid.FacetGrid at 0x1346ae7d6a0>

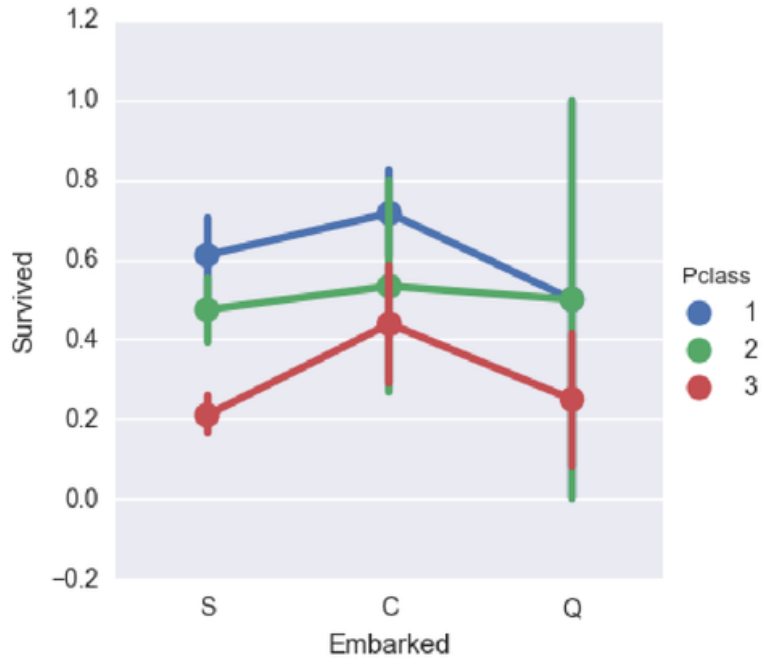


In this figure, the survival rate had similar pattern when we did not separate them by gender. As we can see, the survival rate of female was much higher than male; even female in the third Pclass had higher survival rate thant men in Pclass 1. Moreover, the survival rate of female in Pclass 1 had almost reached to 1 which shows that almost all of the women in Pclass were saved.

After looking at the survival rate by gender and Pclass, is it possible for us to see that did people embarked from different place have different survival rate?

In [35]:
```
# Let's look at how survival rates differ in terms of Embarked and Pclass
sns.factorplot('Embarked', 'Survived', hue = 'Pclass', data = titanicDF)
```

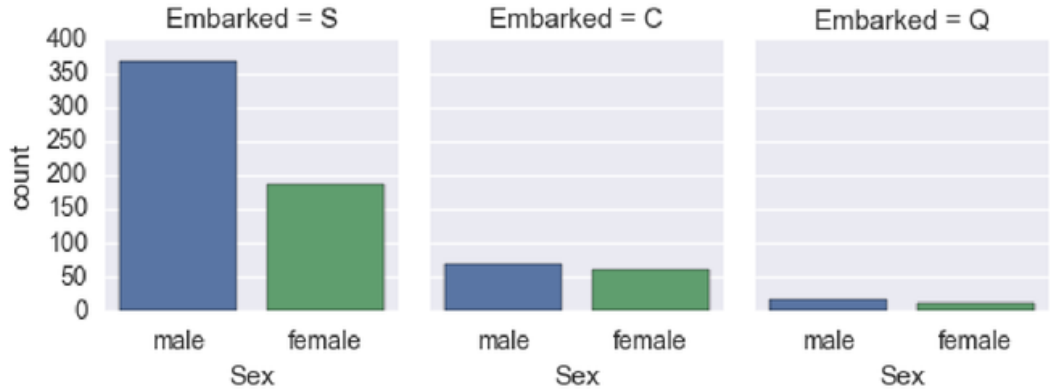
Out[35]: <seaborn.axisgrid.FacetGrid at 0x1346d3d5160>



Surprsingly, it doesn't matter which Pclass were you in, people embarked from *Cherbourg* highest survival rate than the other two places. We would like to ask, is it because there were more women embarked from *Cherbourg*?

In [36]:
```
# Let's look at how survival rates differ in terms of Embarked and Pclass
sns.factorplot('Sex', col='Embarked', data=titanicDF, kind="count", size=2.5, aspect=.8)
```

Out[36]: <seaborn.axisgrid.FacetGrid at 0x1346d8475c0>

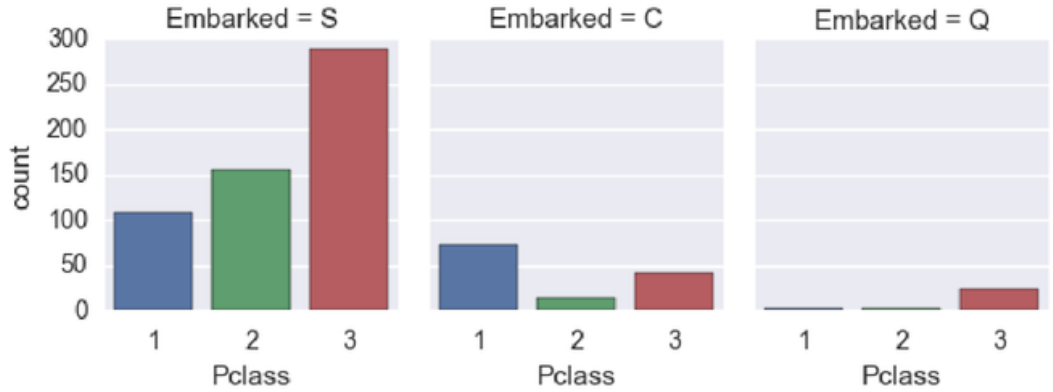


It seems that most of the women embarked from Southampton, but there were far more men embarked from there. Moreover, there were almost equal number of people embarked from Queenstown; as a consequence, we cannot say that it is because there were more women embarked from Cherbourg, so the saved rate of people embarking from Cherborgh had the highest rate.

But about the relation between Pclass and the embarked place?

In [37]:
```
# The relatonship between Pclass and Embarked
sns.factorplot('Pclass', col='Embarked', data=titanicDF, kind="count", size=2.5, aspect=.8)
```

Out[37]: <seaborn.axisgrid.FacetGrid at 0x1346d845ef0>



In [38]:
```
# The relatonship between Pclass and Embarked
sns.factorplot('Pclass', row ='Sex', col= 'Embarked',data=titanicDF, kind="count", size=3, aspect=.8)
```

Out[38]: <seaborn.axisgrid.FacetGrid at 0x1346d3b8a90>

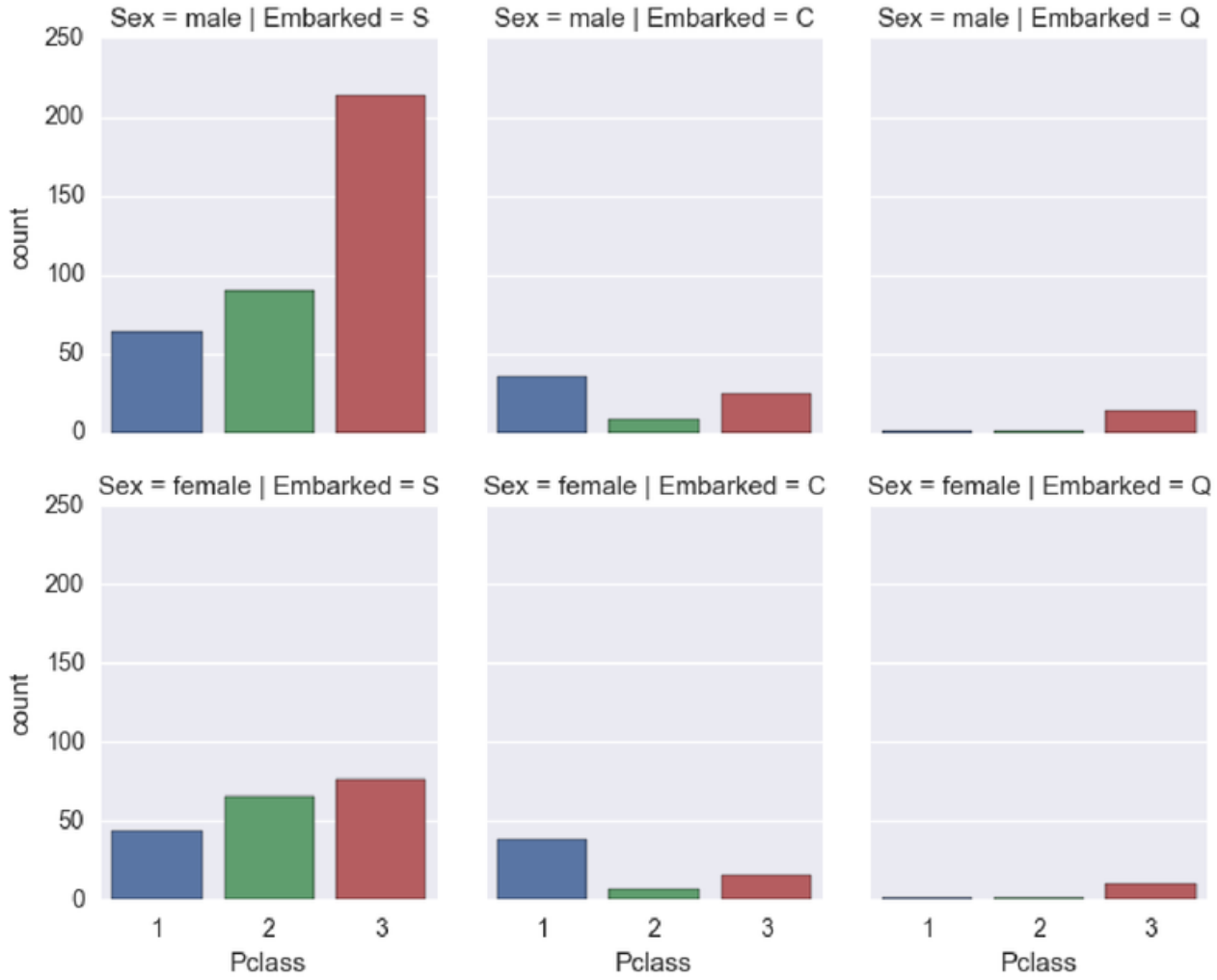
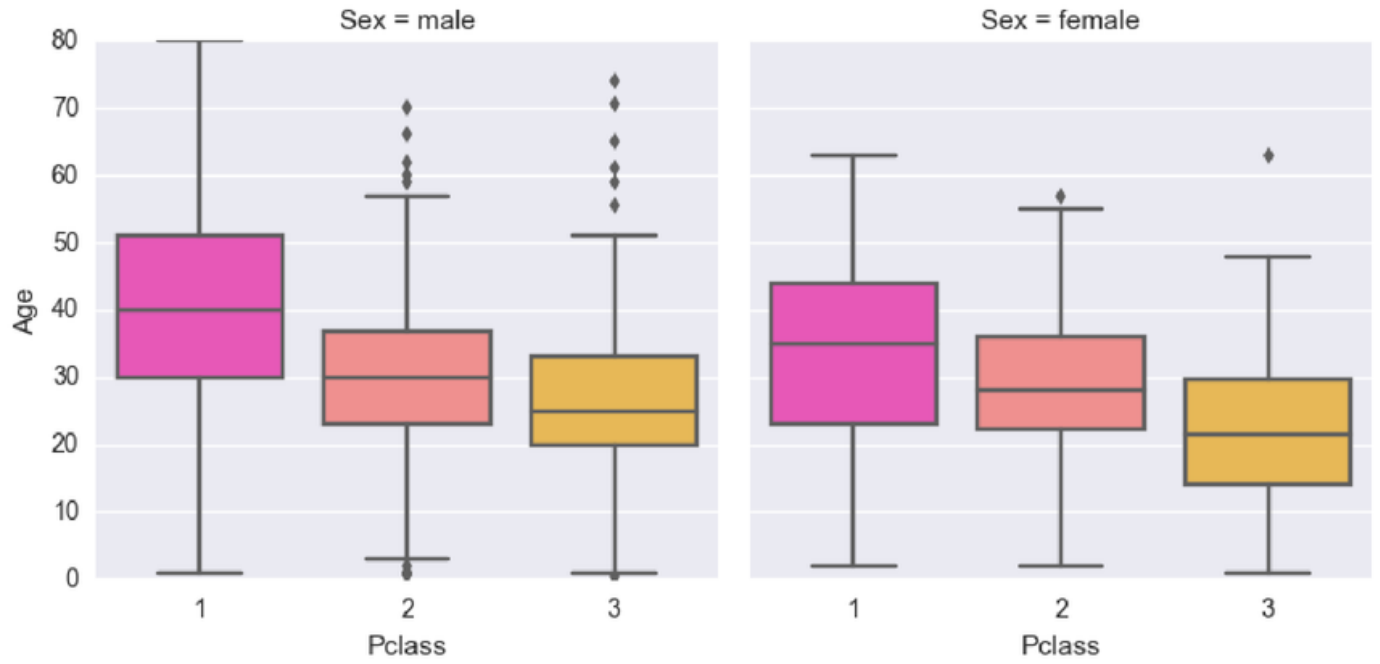


Figure above shows the relationship between Pclass, Embarked Place and Sex.

C. Other Features

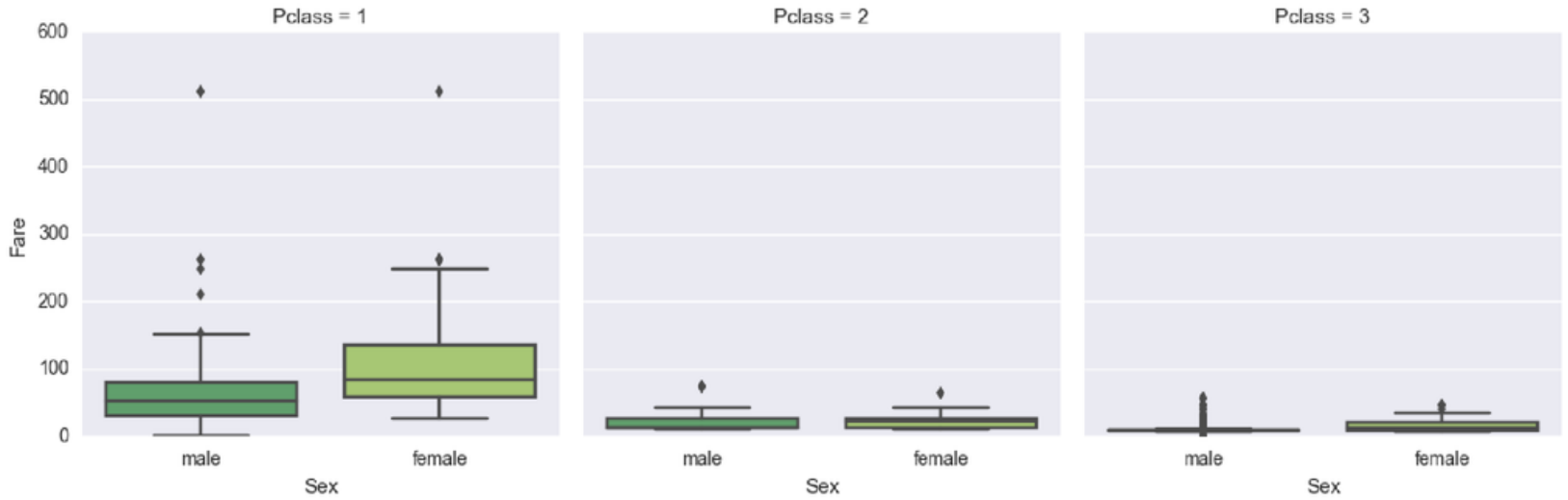
In [39]:
```
# Relationship between Pclass, Age and Sex
sns.factorplot('Pclass','Age', col ='Sex', data = titanicDF, kind="box", palette = 'spring')
```

Out[39]: <seaborn.axisgrid.FacetGrid at 0x1346d3cde80>



In [40]:
```
# Relationship between Pclass, Fare and Sex
sns.factorplot('Sex','Fare',col = 'Pclass', data = titanicDF, kind="box", palette = 'summer')
```

Out[40]: <seaborn.axisgrid.FacetGrid at 0x1346ec0f550>



It seems that female had higher fare price compared to men.

Type *Markdown* and LaTeX: α^2

3. Missing Values

How to impute missing values and what shall we impute are issues for statiscians. At here, we are going to insert the mean of each variable.

Reference: Imputation (statistics (https://en.wikipedia.org/wiki/Imputation_(statistics)

In [41]:
```
features = ["Sex", "Pclass", "SibSp", "Embarked", "Age", "Fare"]
svmDF[features] = svmDF[features].fillna((svmDF[features].mean()), inplace=True)
svmDF.head(6)
# the age of the 6th rows [index = 5] has missing value, so I inspect whether the commend works
```

```
C:\Users\user\Anaconda3\lib\site-packages\pandas\core\generic.py:3191: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: http://pandas.pydata.org/pandas-docs/stable/indexing.html#indexing-view-versus-co
py (http://pandas.pydata.org/pandas-docs/stable/indexing.html#indexing-view-versus-copy)
  self._update_inplace(new_data)
```

Out[41]:

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
0	1	0	3	Braund, Mr. Owen Harris	male	22.000000	1	0	A/5 21171	7.2500	NaN	S
1	2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.000000	1	0	PC 17599	71.2833	C85	C
2	3	1	3	Heikkinen, Miss. Laina	female	26.000000	0	0	STON/O2. 3101282	7.9250	NaN	S
3	4	1	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.000000	1	0	113803	53.1000	C123	S
4	5	0	3	Allen, Mr. William Henry	male	35.000000	0	0	373450	8.0500	NaN	S
5	6	0	3	Moran, Mr. James	male	29.699118	0	0	330877	8.4583	NaN	Q

Type *Markdown* and LaTeX: α^2

4. SVM with Cross-Validation

There are 9 steps in this section: first of all, we need to load the packages and tidy the dataframe. Followed by that, we will spliting data into training and test set. Choose estimator

(1) Loading packages and tidying dataframe

In this section, we will use the same dataframe as in the previous section: **svmDF**. Since we already imputed the mean into missing values and drop the columns we don't need, we only need to change Sex and Embarked into numbers.

Because there is a problem (http://stackoverflow.com/questions/20625582/how-to-deal-with-settingwithcopywarning-in-pandas), so I added *pd.options.mode.chained_assignment = None*

In [42]:
```
# Importing the package
from sklearn import preprocessing, svm, metrics
from sklearn.svm import SVC
```

In [43]:
```
pd.options.mode.chained_assignment = None  # default='warn'
```

In [44]:
```
# Convert the male and female groups to integer form
svmDF["Sex"][svmDF["Sex"] == "male"] = 0
svmDF["Sex"][svmDF["Sex"] == "female"] = 1

# Convert the Embarked classes to integer form
svmDF["Embarked"][svmDF["Embarked"] == "S"] = 0
svmDF["Embarked"][svmDF["Embarked"] == "C"] = 1
svmDF["Embarked"][svmDF["Embarked"] == "Q"] = 2
```

In [45]:
```
svmDF.head(6)
```

Out[45]:

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
0	1	0	3	Braund, Mr. Owen Harris	0	22.000000	1	0	A/5 21171	7.2500	NaN	0
1	2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	1	38.000000	1	0	PC 17599	71.2833	C85	1
2	3	1	3	Heikkinen, Miss. Laina	1	26.000000	0	0	STON/O2. 3101282	7.9250	NaN	0
3	4	1	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	1	35.000000	1	0	113803	53.1000	C123	0
4	5	0	3	Allen, Mr. William Henry	0	35.000000	0	0	373450	8.0500	NaN	0
5	6	0	3	Moran, Mr. James	0	29.699118	0	0	330877	8.4583	NaN	2

In [46]:
```
svmDF = svmDF.drop(['Ticket','Cabin','Name'], axis=1).dropna() ; svmDF.head()
```

Out[46]:

	PassengerId	Survived	Pclass	Sex	Age	SibSp	Parch	Fare	Embarked
0	1	0	3	0	22.0	1	0	7.2500	0
1	2	1	1	1	38.0	1	0	71.2833	1
2	3	1	3	1	26.0	0	0	7.9250	0
3	4	1	1	1	35.0	1	0	53.1000	0
4	5	0	3	0	35.0	0	0	8.0500	0

(2) Spliting data into training and test set

Document (http://scikit-learn.org/stable/modules/cross_validation.html) of sklearn

In [47]:
```
try:
    from sklearn.model_selection import train_test_split, cross_val_score
except ImportError:
    from sklearn.cross_validation import train_test_split, cross_val_score
```

In [48]:
```
'''
## Reminder:
features = ["Sex", "Pclass", "SibSp", "Embarked", "Age", "Fare"]
'''

# titanicDF without 'Survived'
titanicX = svmDF[features]

# array of 'Survived'
titanicy = svmDF['Survived']
```

In [49]:
```
titanicX.isnull().values
```

Out[49]:
```
array([[False, False, False, False, False, False],
       [False, False, False, False, False, False],
       [False, False, False, False, False, False],
       ...,
       [False, False, False, False, False, False],
       [False, False, False, False, False, False],
       [False, False, False, False, False, False]], dtype=bool)
```

In [50]:
```
titanicy.isnull().values.any()
```

Out[50]:
```
False
```

In [51]:
```
X_train, X_test, y_train, y_test = train_test_split(\
        titanicX, titanicy, test_size=0.4, random_state=0)
```

(3) Choose estimator

In [52]:
```
# Train the model
clf = svm.SVC(kernel='linear')
```

In [53]:
```
clf.fit(X = X_train, y = y_train)
```

Out[53]:
```
SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0,
  decision_function_shape=None, degree=3, gamma='auto', kernel='linear',
  max_iter=-1, probability=False, random_state=None, shrinking=True,
  tol=0.001, verbose=False)
```

In [54]:
```
# get support vectors
print(clf.support_vectors_)
```

```
[[  0.          1.          0.          0.         29.69911765   0.        ]
 [  0.          2.          0.          0.         27.          13.        ]
 [  1.          3.          0.          0.         25.           7.775     ]
 ...,
 [  0.          1.          1.          0.         27.          53.1       ]
 [  0.          2.          1.          0.          3.          26.        ]
 [  0.          1.          0.          0.         80.          30.        ]]
```

It has some problems, because there are too many points. The possible reason is because of the distribution of the data.

In [55]:
```
# get indices of support vectors
print(clf.support_ )
```

```
[  6   7  13  18  31  37  38  39  45  47  56  59  61  64  65  66  76  88
  92  98 100 103 105 108 111 117 128 135 136 141 143 146 149 159 162 173
 176 179 180 185 189 195 199 210 213 219 236 237 239 243 247 249 255 257
 258 263 269 270 273 274 283 291 296 303 313 315 316 317 321 322 324 330
 332 344 345 348 351 357 359 365 369 370 374 378 380 383 384 396 407 411
 420 421 427 433 435 436 438 439 440 447 452 456 467 469 480 482 493 494
 495 506 518   0  12  16  17  21  22  41  42  43  46  51  55  58  71  78
  80  84  87  91  95  99 104 109 110 115 116 122 124 126 129 132 138 140
 142 150 151 156 165 171 172 177 178 183 190 198 200 209 212 224 230 232
 233 238 250 254 259 260 262 267 272 275 280 282 287 295 298 299 300 302
 307 319 323 333 334 337 338 339 350 356 358 360 367 379 385 386 388 390
 393 401 410 412 414 424 445 446 449 453 463 468 470 473 476 490 496 497
 510 513 514 519 521 523 529 530]
```

In [56]:
```
# get number of support vectors for each class
print(clf.n_support_)
```

```
[111 113]
```

(4) Choose cross-validation iterator

Reference:

- Cross-validation (http://scikit-learn.org/stable/modules/cross_validation.html)
- The scoring parameter (http://scikit-learn.org/stable/modules/model_evaluation.html#scoring-parameter)

In [57]:
```
from sklearn.cross_validation import ShuffleSplit

cv = ShuffleSplit(X_train.shape[0], n_iter=10, test_size=0.2, random_state=0)
```

```
C:\Users\user\Anaconda3\lib\site-packages\sklearn\cross_validation.py:44: DeprecationWarning: This module was deprecate
d in version 0.18 in favor of the model_selection module into which all the refactored classes and functions are moved.
 Also note that the interface of the new CV iterators are different from that of this module. This module will be remov
ed in 0.20.
  "This module will be removed in 0.20.", DeprecationWarning)
```

(5) Tuning the hyperparameters

Reference: Tuning the hyper-parameters of an estimator (http://scikit-learn.org/stable/modules/grid_search.html#grid-search)

In [58]:
```
clf.get_params()
```

Out[58]:
```
{'C': 1.0,
 'cache_size': 200,
 'class_weight': None,
 'coef0': 0.0,
 'decision_function_shape': None,
 'degree': 3,
 'gamma': 'auto',
 'kernel': 'linear',
 'max_iter': -1,
 'probability': False,
 'random_state': None,
 'shrinking': True,
 'tol': 0.001,
 'verbose': False}
```

I need to add words.

Otherwise,

the rest of my code will disappear. Therfore I need to add words. Therfore I need to add words. Therfore I need to add words. Therfore I need to add words. Therfore I need to add words. Therfore I need to add words. Therfore I need to add words. Therfore I need to add words. Therfore I need to add words. Therfore I need to add words. Therfore I need to add words. Therfore I need to add words. Therfore I need to add words. Therfore I need to add words. Therfore I need to add words. Therfore I need to add words.Therfore I need to add words.

In [ ]:
```
It seems that adding a markdown cell do work.
```

GridSearchCV (Super Slow!!!)

Reference: GridSearch (http://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html#sklearn.model_selection.GridSearchCV)

In [64]:
```
from sklearn.grid_search import GridSearchCV
param_grids = [
  {'C': [1, 10, 100], 'kernel': ['linear']},
 ]
classifier = GridSearchCV(estimator=clf, param_grid=param_grids, cv=cv)
classifier.fit(X_train, y_train)
```

Out[64]:
```
GridSearchCV(cv=ShuffleSplit(533, n_iter=10, test_size=0.2, random_state=0),
       error_score='raise',
       estimator=SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0,
  decision_function_shape=None, degree=3, gamma='auto', kernel='linear',
  max_iter=-1, probability=False, random_state=None, shrinking=True,
  tol=0.001, verbose=False),
       fit_params={}, iid=True, n_jobs=1,
       param_grid=[{'C': [1, 10, 100], 'kernel': ['linear']}],
       pre_dispatch='2*n_jobs', refit=True, scoring=None, verbose=0)
```

(6) Final evaluation on the test set

In [68]:
```
classifier.score(X_test, y_test)
```

Out[68]:
```
0.7696629213483146
```

In [69]:
```
cross_val_score(clf, X_train, y_train, cv=cv, scoring='f1')
```

Out[69]:
```
array([ 0.73913043,  0.72131148,  0.73170732,  0.74074074,  0.69333333,
        0.60606061,  0.66666667,  0.73239437,  0.73170732,  0.71604938])
```

the rest of my code will disappear. Therfore I need to add words. Therfore I need to add words. Therfore I need to add words. Therfore I need to add words. Therfore I need to add words. Therfore I need to add words. Therfore I need to add words. Therfore I need to add words. Therfore I need to add words. Therfore I need to add words. Therfore I need to add words. Therfore I need to add words. Therfore I need to add words. Therfore I need to add words.Therfore I need to add words.

In [ ]:
```

```

(7) Training final model on whole dataset

In [ ]:
```
classifier.fit(titanicX, titanicy)
```

Conclusion

We have sorted the feature importance and found that themost important feature for predicting survived or not is 'Age', followed by 'Fare', 'Sex', 'Pclass', 'Sibsp' , and 'Embarked'. After dropping the missing values, we got 712 number of passengers' data with 288 saved and 424 people died.

-***Age***: We found that most of the passangers were aged between 20 to 30 year old, but people in different Pclass had sligtly different age distribution. People in Pclass 3 were younger than the other two on average. While more people from Plcass 2 and Pclass 3 aged between 20 to 30 year old, there were more people aged 30 to 40 year old with Pclass1; also, the age of Plcass 1 were more evenly distributed comparing to other two groups.

-***Pclass***: We found that there were more people survived in the Pclass 1 than the people in the Pclass 3. In addition, the survival rate of female was much higher than male; even female in the third Pclass had higher survival rate thant men in Pclass1.

-***Sex***: 20% of men died but 75% of women survived.

-***Fare***: It seems that female had higher fare price compared to men.

-***Embarked***: Surprsingly, it doesn't matter which Pclass were you in, people embarked from Cherbourg highest survival rate than the other two places.

-***SVM and Cross-Validation***: Becuase there are a lot of points on the decision bundary, it seems that using SVM with linear kernel is not a good classifier to predict whether a person would save or died by the given data.

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]: