In [1]:

```
import numpy as np
import pandas as pd
from bokeh.charts import Scatter
from bokeh.plotting import Figure
from bokeh.models import Span
from bokeh.io import output_notebook, show
output_notebook()

from sklearn.model_selection import train_test_split
#sk런은 사이킷을 지칭하는 용어이다.  scikit-learn, 선형회귀를 위한 파이썬 라이브러리?
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error
```

(http://bokeh.pydata.org)

BokehJS successfully loaded.

Linear Regression

As discussed in the videos, one part of both statistics and machine learning is linear regression. There are several Python libraries for this as well. The one we'll be using in this course is the **scikit-learn LinearRegression class (http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LinearRegression.html#sklearn.linear_model.Li** which uses the **ordinary least squares** method to calculate predictions.

We will perform linear regression on one of the sample data sets shipped with Bokeh: the "Iris" dataset, which consists of data about flowers. First, we import and make a copy of the `flowers` dataset, and display the fields associated with it to get an idea of the data:

In [11]:

```
# Creating a DataFrame using the sample data provided by Bokeh
from bokeh.sampledata.iris import flowers
flowers_data = flowers

# Displaying first few rows of data
#flowers_data.head()
print(flowers_data)
print(type(flowers_data))
```

```
     sepal_length  sepal_width  petal_length  petal_width    species
0             5.1          3.5           1.4          0.2     setosa
1             4.9          3.0           1.4          0.2     setosa
2             4.7          3.2           1.3          0.2     setosa
3             4.6          3.1           1.5          0.2     setosa
4             5.0          3.6           1.4          0.2     setosa
5             5.4          3.9           1.7          0.4     setosa
6             4.6          3.4           1.4          0.3     setosa
7             5.0          3.4           1.5          0.2     setosa
8             4.4          2.9           1.4          0.2     setosa
9             4.9          3.1           1.5          0.1     setosa
10            5.4          3.7           1.5          0.2     setosa
11            4.8          3.4           1.6          0.2     setosa
12            4.8          3.0           1.4          0.1     setosa
13            4.3          3.0           1.1          0.1     setosa
14            5.8          4.0           1.2          0.2     setosa
15            5.7          4.4           1.5          0.4     setosa
16            5.4          3.9           1.3          0.4     setosa
17            5.1          3.5           1.4          0.3     setosa
18            5.7          3.8           1.7          0.3     setosa
19            5.1          3.8           1.5          0.3     setosa
20            5.4          3.4           1.7          0.2     setosa
21            5.1          3.7           1.5          0.4     setosa
22            4.6          3.6           1.0          0.2     setosa
23            5.1          3.3           1.7          0.5     setosa
24            4.8          3.4           1.9          0.2     setosa
25            5.0          3.0           1.6          0.2     setosa
26            5.0          3.4           1.6          0.4     setosa
27            5.2          3.5           1.5          0.2     setosa
28            5.2          3.4           1.4          0.2     setosa
29            4.7          3.2           1.6          0.2     setosa
..            ...          ...           ...          ...        ...
120           6.9          3.2           5.7          2.3  virginica
121           5.6          2.8           4.9          2.0  virginica
122           7.7          2.8           6.7          2.0  virginica
123           6.3          2.7           4.9          1.8  virginica
124           6.7          3.3           5.7          2.1  virginica
125           7.2          3.2           6.0          1.8  virginica
126           6.2          2.8           4.8          1.8  virginica
127           6.1          3.0           4.9          1.8  virginica
128           6.4          2.8           5.6          2.1  virginica
129           7.2          3.0           5.8          1.6  virginica
130           7.4          2.8           6.1          1.9  virginica
131           7.9          3.8           6.4          2.0  virginica
132           6.4          2.8           5.6          2.2  virginica
133           6.3          2.8           5.1          1.5  virginica
134           6.1          2.6           5.6          1.4  virginica
135           7.7          3.0           6.1          2.3  virginica
136           6.3          3.4           5.6          2.4  virginica
137           6.4          3.1           5.5          1.8  virginica
```

```
138           6.0           3.0           4.8           1.8  virginica
139           6.9           3.1           5.4           2.1  virginica
140           6.7           3.1           5.6           2.4  virginica
141           6.9           3.1           5.1           2.3  virginica
142           5.8           2.7           5.1           1.9  virginica
143           6.8           3.2           5.9           2.3  virginica
144           6.7           3.3           5.7           2.5  virginica
145           6.7           3.0           5.2           2.3  virginica
146           6.3           2.5           5.0           1.9  virginica
147           6.5           3.0           5.2           2.0  virginica
148           6.2           3.4           5.4           2.3  virginica
149           5.9           3.0           5.1           1.8  virginica

[150 rows x 5 columns]
<class 'pandas.core.frame.DataFrame'>
```

We are going to attempt to see **what features are best used** to calculate the sepal (https://en.wikipedia.org/wiki/Sepal). We start by creating an instance of the `LinearRegression` class

In [7]:

```
# lm is an instance of the LinearRegression class
lm = LinearRegression() # 실제로 LinearRegression()클래스를 결합
```

Fitting the Model

We are going to begin by creating a model using the petal length (as the **predictor variable**) to calculate the sepal length (the **response** variable).

We need some X and y values, which will be used to call the **`fit`** function. Create a variable X for predictor variables, and a variable y for the associated response variable. These will be data frames taken from the `flowers_data` variable.

In [14]:

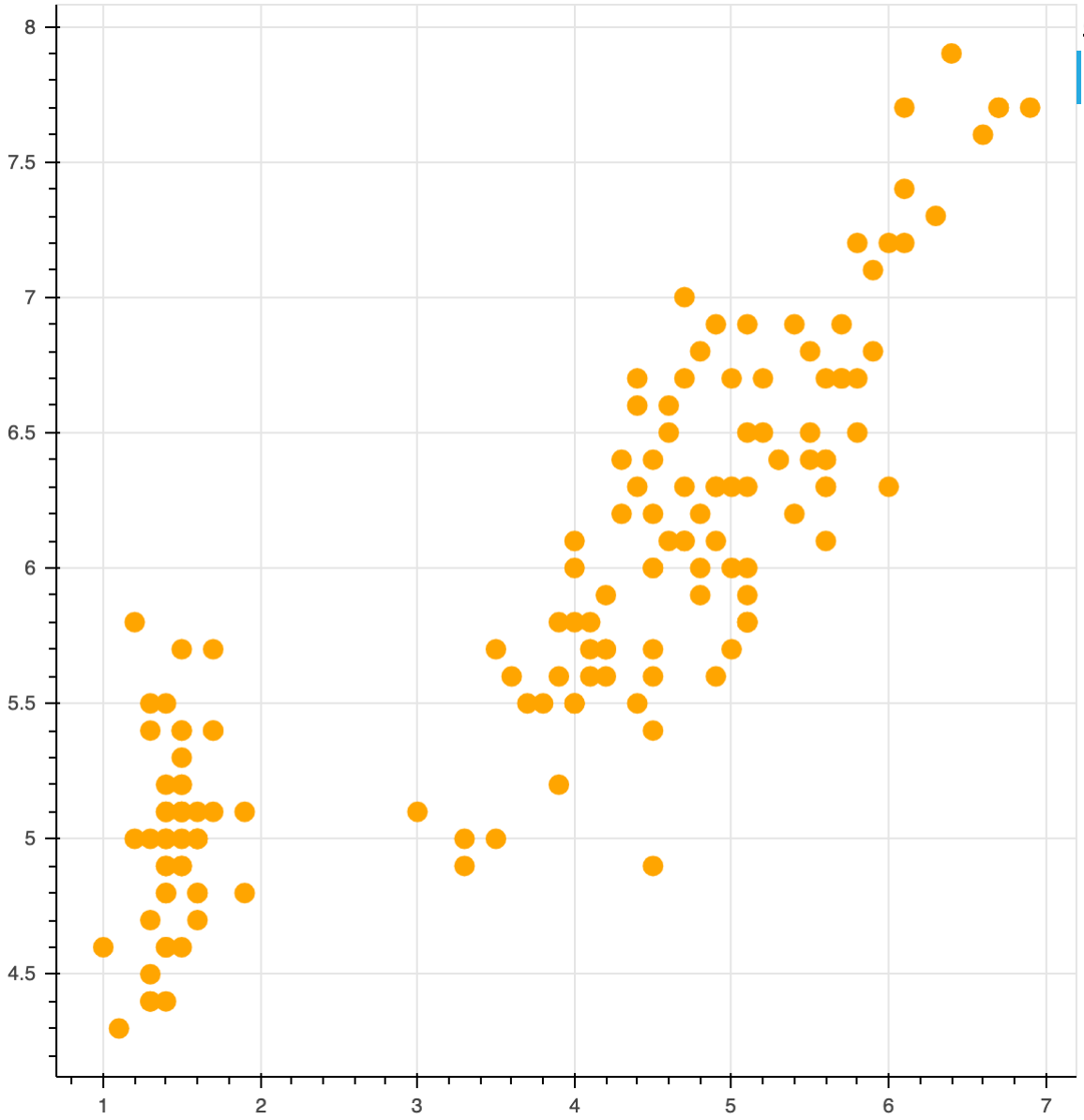
```
X = flowers_data[['petal_length']] #꽃잎 길이 = 예측 변수 (예측해서 사용을 하면)
#한개 이상의 열이 있을지도 모르기에 두 쌍의 괄호를 사용했음
y = flowers_data['sepal_length'] #꽃받침 길이 = 반응 변수 (그에 따라 반응하는 변수)
```

We now use the Bokeh `Figure` class to illustrate the distribution of this predictor:

Follow this link to find out more about Bokeh.plotting (http://bokeh.pydata.org/en/0.11.0/docs/reference/plotting.html).

In [12]:

```
fig = Figure() #시각화 프로그램인 보케를 이용하여 차트를 만들거나 산포 그래프를
#무언가에 할당해야하므로 Figure()를 fig에 저장 !
fig.circle(X['petal_length'],y, size = 10, color = 'orange')
#함수의 특성인 circle에 X값과 y값 할당.  !
show(fig)
```



(http

Having prepared the data, we can now fit the data to a linear regression model, and then begin analysis:

In [11]:

```
# Call the fit function on the data
lm = LinearRegression()
lm.fit(X, y) #fit(X,y)를 선형회귀에 적용 !
print(vars(lm)) #딕셔너리 출력, 전차, 랭크, 계수 등 선형 회귀에 대한 정보 출력
```

```
{'copy_X': True, 'fit_intercept': True, 'singular_': array([ 21.548211
06]), 'normalize': False, 'intercept_': 4.3066034150475794, 'n_jobs':
1, 'coef_': array([ 0.40892228]), '_residues': 24.525033765831754, 'ra
nk_': 1}
```

Having fitted the model to the data, we can establish what the regression line by using the `coef_` and `intercept` values, which we can use in order to generate an equation in the format `y = mx + c`:

In [12]:

```
m = lm.coef_[0] #coef_, intercept_ 둘 다 선형 회귀 라이브러리의 함수임 !
c = lm.intercept_
print('y = %fx + %f' % (m,c))
```

```
y = 0.408922x + 4.306603
```

We don't necessarily need to use the equation, however, there is additionally a useful **`predict`** function which allows us to enter one or more numbers for which it will predict the outcome.

In [13]:

```
#예측변수 살펴보기

var = 0

print(lm.predict([var])) #예측 변수 함수를 변수 var에 적용해서 출력

# Note that like with the earlier Series, the `predict` function is prepared for the
# to predict could have multiple features.  So it accepts a list of lists
# This will give a deprecation warning
var = [[3], [5], [4]]
lm.predict(var)

# if a warning message box appears this indicates a possible depracated code issue,
# you have not broken the internet!
```

```
[ 4.30660342]

/opt/conda/lib/python3.5/site-packages/sklearn/utils/validation.py:39
5: DeprecationWarning: Passing 1d arrays as data is deprecated in 0.17
and will raise ValueError in 0.19. Reshape your data either using X.re
shape(-1, 1) if your data has a single feature or X.reshape(1, -1) if
it contains a single sample.
  DeprecationWarning)
```

Out[13]:

```
array([ 5.53337025,  6.3512148 ,  5.94229252])
```

Evaluation of a Model

Training and Testing Data

In order to ensure that our model can cope with the whole of the dataset, we will split our data into a **training set** and a **testing set**, and **cross validate** the results. We need to be careful that the ordering of our data is sufficiently **random** when doing this. For example, if it was sorted by value, we would be training only on a specific type of value making our model less accurate against values outside that range.

The function **'train_test_split' (http://scikit-learn.org/stable/modules/generated/sklearn.model_selection.train_test_split.html)** from the `sklearn.model_selection` module performs this task for us - both splitting the data, and ensuring that the ordering is sufficiently random. It returns four values, we will assign each of those a value as the output of this function by declaring them on the same line as follows:

In [20]:

```
# The function 'train_test_split' 을 사용 !
# 데이터를 분할하고 순서가 충분히 무작위인지 확인하는 작업을 저 함수가 수행 !

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)
#총 네가지 값이 있음. 이 결과값이 각각에 할당 됨
print(X_test)
```

```
     petal_length
81            3.7
29            1.6
94            4.2
143           5.9
8             1.4
103           5.6
51            4.5
32            1.5
26            1.6
65            4.4
14            1.2
22            1.0
96            4.2
88            4.1
109           6.1
133           5.1
27            1.5
5             1.7
123           4.9
2             1.3
70            4.8
7             1.5
136           5.6
20            1.7
64            3.6
1             1.4
46            1.6
21            1.5
58            4.6
126           4.8
```

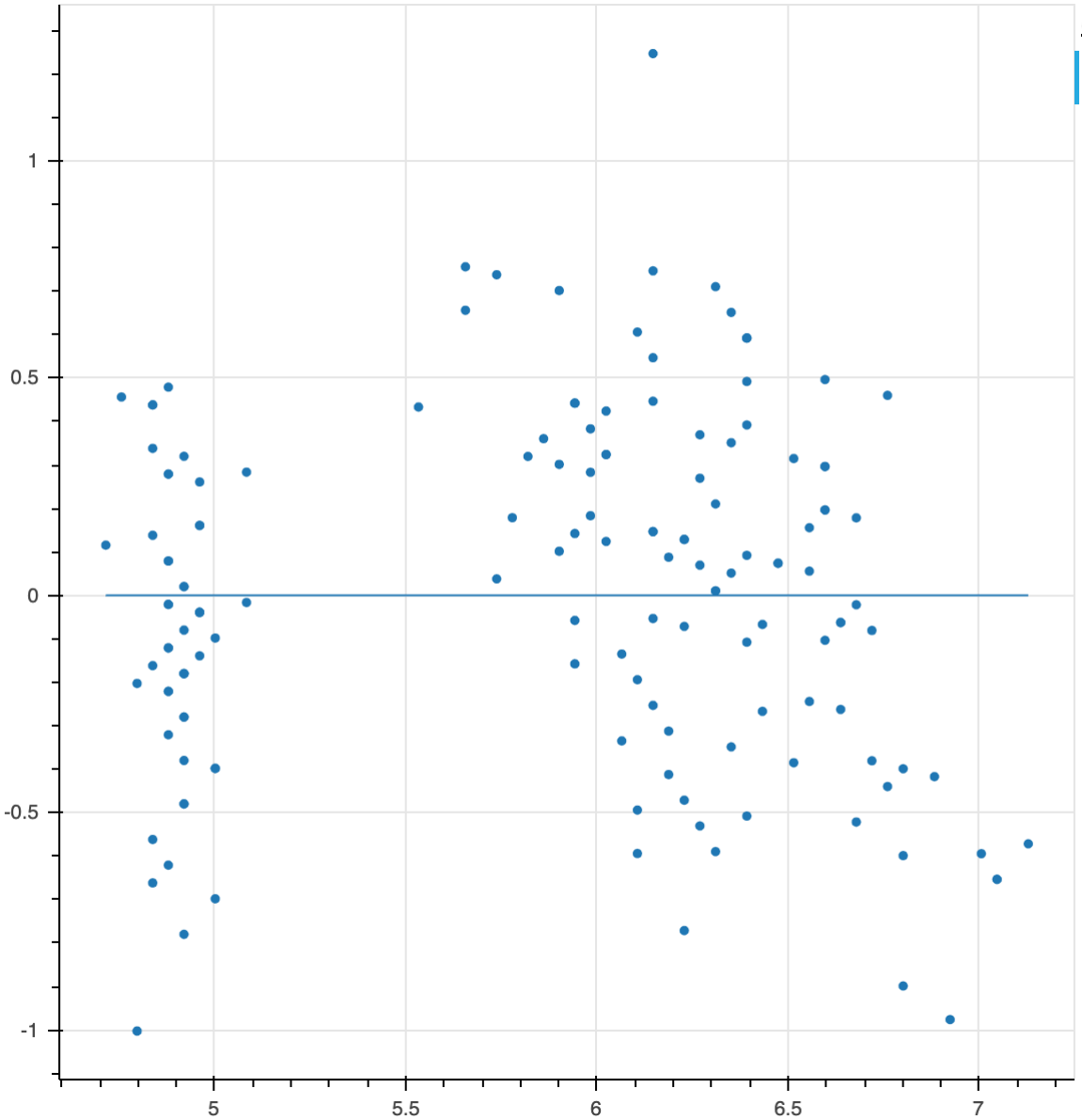
Residual Analysis

In order to ensure that our model is appropriate for the data, we first make an analysis of the residuals, i.e., the difference between the predicted value and the observed value. If the data are randomly clustered around 0, then that is a sign the data are appropriate. However, if there is a pattern in the data, then that suggests that there is some form of bias.

We can use the `predict` method of the `LinearRegression` class to get the predicted values (which we'll use as the `X` values), and we subtract the actual values from them, as follows:

In [15]:

```
pred_values = lm.predict(X)
pred_difference = (pred_values - y) # (예측값을 가져오고 , 실제값을 뺍니다)
fig = Figure()
fig.circle(pred_values,pred_difference)
fig.line(x=[pred_values.min(), pred_values.max()], y=[0,0])
show(fig)
```



(http

R^2

The R^2 value provides us with the **proportion of variance** which is explained by the model. It is a value between 0 and 1, with 0 being that the model has no explanation for any of the variance and 1 being that the regression line fits the data perfectly.

The defined range of R^2 means that it can give a reasonable indication of how good a model is, although there is no defined range of what constitutes a good enough score. This would naturally vary between domains, where there is a smaller margin of error than others.

In scikit-learn, the **`score`** method calculates the R^2 value as follows:

In [16]:

```
#잔차 제곱 사용
lm.score(X, y) #scikit에 score로 R제곱을 계싼합니다.
#0는 어떤 분산도 설명하지 않고, 1은 회귀가 데이터에 완벽하게 부합한다고 함
#0.75995464577251504, 따라서 이 데이터가 편향되지 않았다는 것을 상대적으로 확신 할 수 있음 !
```

Out[16]:

```
0.75995464577251504
```

RMSE

We now use **RMSE** (Root Mean Squared Error) to **evaluate** our model. Whereas the residual analysis considers the individual residual difference, RMSE takes the mean of these residuals squared and takes the square root of the total. This output is the **standard deviation** of a model, and can be used as a measure of accuracy.

Further information about mean_squared_error (http://scikit-learn.org/stable/modules/generated/sklearn.metrics.mean_squared_error.html) and square root (https://docs.scipy.org/doc/numpy/reference/generated/numpy.sqrt.html)

There is a function for calculating the mean squared error, and then a numpy function to calculate the square root of this. We test the ***actual*** data against the ***predicted*** data performed in the previous cell as follows:

In [21]:

```
# RMSE : 평균 제곱근 편차

np.sqrt(mean_squared_error(X, y))
```

Out[21]:

```
2.365079279855117
```

By default, the `test_size` argument is 0.25, which means that when it comes to validating the model, we test on a quarter of the data, having trained our model on the complement (three quarters). Note the difference in size between the training and testing set, and the ordering of the indexes:

In [22]:

```
# 트레이닝 세트 활용, 전체 데이터 세트를 활용해서 모델을 구축하고 싶지 않을 때 사용
#일정한 양으로 테스트를 진행


print('Training data: %d\n' % X_train.size, X_train.head(), '\nTest data: %d\n' % X_
#보통 Test data는 Training data의 1/4 크기이다. (120개/30개)
```

```
Training data: 120
      petal_length
108            5.8
117            6.7
3              1.5
4              1.4
17             1.4
Test data: 30
      petal_length
81             3.7
29             1.6
94             4.2
143            5.9
8              1.4
```

Our model gets trained as follows:

In [23]:

```
lm = LinearRegression()
lm.fit(X_train, y_train)
y_pred = lm.predict(X_test) #30개 꽃잎 길이 테스트 값을 사용
#y의 예측값을 얻기위해 y_pred 사용하고,  X 데이터에 대한 predict 함수를 사용
#실제 X꽃잎길이 가 있을 경우에 예측된 Y 꽃받침 길이 세트를 반환
print(y_pred)
```

```
[ 5.81066108  4.92696373  6.02106521  6.73643925  4.84280207  6.610196
77
  6.14730769  4.8848829   4.92696373  6.10522686  4.75864042  4.674478
77
  6.02106521  5.97898438  6.8206009   6.39979264  4.8848829   4.969044
55
  6.31563099  4.80072125  6.27355016  4.8848829   6.61019677  4.969044
55
  5.76858025  4.84280207  4.92696373  4.8848829   6.18938851  6.273550
16]
```

Try it yourself...

There are two more features which you can use to predict the sepal length of a flower. Try and find the best combination of predictor variables in the cells below. If you need to create a new cell, click on `Insert -> Insert Cell Below`

In [ ]:

```
# YOUR CODE HERE
```

In [ ]:

```
# YOUR CODE HERE
```

Classification

To finish, we will walk through an example classification with a spam filter. The process with scikit-learn is similar to the one described for linear regression.

We won't go into too much detail on classification, which is the other form of supervised machine learning. However in this notebook, we will present an example of using a **Bayesian classifier** to classify whether SMS text messages are "spam" or "ham".

We use the dataset from UCI Machine Learning Repository (https://archive.ics.uci.edu/ml/machine-learning-databases/00228/). These data are stored in the same directory as the notebook in spam.csv (./spam.csv)

To start, we import the libraries we require to perform this classification, and load the data into a `DataFrame`. Note that although it's a CSV file, we specify `sep` value as being `'\t'`, which means that the two fields are separated by a tab character rather than a comma.

In [3]:

```
import numpy as np
import pandas as pd
import sklearn
import sklearn.naive_bayes as nb
import sklearn.feature_extraction.text as text
import sklearn.model_selection as cv
df = pd.read_csv('spam.csv', sep='\t')
df.head()
```

Out[3]:

	Classification	Message
0	ham	Go until jurong point, crazy.. Available only ...
1	ham	Ok lar... Joking wif u oni...
2	spam	Free entry in 2 a wkly comp to win FA Cup fina...
3	ham	U dun say so early hor... U c already then say...
4	ham	Nah I don't think he goes to usf, he lives aro...

In order to be able to get the features, we need to identify features (i.e., the words) which are classified in a particular way. We need to get some measure of the way that the words are distributed within the messages, and use that to predict the classification into Ham or Spam.

There are several ways of assigning the weight to a particular word. tf-idf (https://en.wikipedia.org/wiki/Tf%E2%80%93idf) is an information retrieval technique to identify the importance of words in a document based on the times a word is in an individual message, but taking account for the times it appears in the whole document - since some words generally appear more often than others.

The "bag of words" (https://en.wikipedia.org/wiki/Bag-of-words_model) is simple: it simply ignores grammar and makes a count of the times that a particular word appears. This is a commonly used approach, which we will use below.

Scikit-learn allows a particular `Vectorizer` (in this instance CountVectorizer (http://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.CountVectorizer.html)) to be selected to implement one of these strategies, as follows:

In [4]:

```
counts = text.CountVectorizer() ##bag of words 메소드를 사용하려면 백터라이저가 필요
#그 중 하나 선택 하여 적용 ( CountVec~)
#실행을 하면 counts가 sk런에서 인식되고 처리될 수 있음
```

For scikit-learn to be able to process the data, it needs the data to be as numbers. The fit_transform (http://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.CountVectorizer.html#sklearn.feature_extract function converts these data, and we convert the classification to a boolean, which Python can consider as an integer.

We define the features as `X` and the labels as `y`

In [5]:

```
# The fit_transform function
X = counts.fit_transform(df['Message'])
# 변수 X값이 df['Message']에 적용된 fit_transform에 할당됨
#fit_transform는 하위함수 혹은 counts의 함수.
print(X)

# Changing the classification field to a boolean, Python can regard it as an integer
y = df['Classification'] == 'spam'
                        #수치상으로 혹은 정확하게 문자열 spam과 일치한다
print(X, y.head())
```

```
  (0, 8324)     1
  (0, 1082)     1
  (0, 3615)     1
  (0, 7694)     1
  (0, 2061)     1
  (0, 1765)     1
  (0, 4501)     1
  (0, 8548)     1
  (0, 3655)     1
  (0, 1767)     1
  (0, 4114)     1
  (0, 5571)     1
  (0, 1316)     1
  (0, 2338)     1
  (0, 5958)     1
  (0, 4374)     1
  (0, 8084)     1
  (0, 3571)     1
  (1, 5567)     1
  (1, 8450)     1
```

As with linear regression, we split the data into a training and a testing set using the `train_test_split` function, and we fit the training data to the classifier.

There are several different types of naive bayes classifier we can use. In this instance, we are using the multinomial naive bayes (http://scikit-learn.org/stable/modules/generated/sklearn.naive_bayes.MultinomialNB.html) classifier:

In [13]:

```
# Split to a training set and a testing set
(text_train, text_test, label_train, label_test) = cv.train_test_split(X, y, test_si

from sklearn.naive_bayes import MultinomialNB

classifier = MultinomialNB() #나이브 베이즈 를 변수 classifier에 할당 .
classifier.fit(text_train, label_train)

#출력값 : 값 0은 스무딩이 없고, 1은 데이터세트에서 최고 스무딩
```

Out[13]:

```
MultinomialNB(alpha=1.0, class_prior=None, fit_prior=True)
```

Having trained the model, we can get a simple test of the accuracy by using the score (http://scikit-learn.org/stable/modules/model_evaluation.html) method:

In [14]:

```
classifier.score(text_test, label_test)

#출력값 : 0.98의 높은 정확도를 의미 , text_test와 lable_test의 예측값이 높은 수준의 정확도를 보임
```

Out[14]:

```
0.98026905829596411
```

We can see that the classifier is already performing very well. Note that it will give a slightly different result each time, because the classifier is trained on different messages as the order is randomised.

The confusion matrix (http://scikit-learn.org/stable/modules/generated/sklearn.metrics.confusion_matrix.html) shows us the breakdown of instances of true positives, true negatives, false positives, and false negatives. There is a method for this as well, used upon the test data.

In [15]:

```
from sklearn.metrics import confusion_matrix, f1_score
                            #혼동 행렬 사용, 얼마나 많은 참 긍정, 거짓 긍정
                            #                    참 부정, 거짓 부정 나오는지 알 수 있음.

predictions = classifier.predict(text_test)
confusion_matrix(label_test, predictions)
```

Out[15]:

```
array([[969,  18],
       [  4, 124]])
```

It is possible to try our own messages as well, and see how those perform. Once again, they need to be transformed into the correct format for scikit-learn. The output is as `False` (not spam) and `True` (spam):

In [16]:

```
messages = ["this is a test", "spam, spam, spam, glorious spam"]

#false면 스팸이 아니고, True 면 스팸이다

transformed_messages = counts.transform(messages)
predictions = classifier.predict(transformed_messages)
print(predictions)
```

```
[False  True]
```

In [ ]:

```

```