Data Analysis/Machine Learning Methods - Mofrad Lab

Vyom Thakkar, Sumiran Singh Thakur

January 2020

Often many machine learning methods are treated as a blackbox, though in many cases certain models fit well while others aren't the appropriate choice. The goal of this guide is to show how a cursory understanding of models and their charecteristics and strengths can allow someone to quickly determine which models are most applicable and which ones aren't suitable to the model at hand. Here we give an introduction to the derivation of each method along with some mathematical intuition behind each process. Alongside each method, we discuss the underlying assumptions and applicable types of data/problems, as well as best practices in evaluating the models and setting hyperparameters. Finally, with application to research in mind, many of the methods also include examples of visualizations. We also contextualize the visuals, with the goal in mind of allowing people to easily refer to the guide for explanations of the methods and the visuals, for ease of use in future research projects in the Mofrad Lab.

Contents

1 Regression

1.1 Ordinary Least Squares

Ordinary least squares represents one of the simplest forms of regression problems, but is however a useful tool if the numerical features of the data (dependent variables) is without noise. Ordinary Least Squares generally assumes that the labels of said data (which are independent variables) are noisy in nature. We then predict weights upon the fetaures, using saif features and labels, and then check to see the error betwen the predicted labels and our labels.

In [1]:
```
import numpy as np
from numpy.linalg import matrix_rank
import matplotlib.pyplot as plt
%matplotlib inline
from sklearn.datasets import make_regression
```

In [2]:
```
#this method is an ordinary least square solvers, good for datasets with uniform
data, no need for reguralization
def OLS(A,b):
    #solving the equation Ax=b
    if (matrix_rank(np.matmul(A.T,A)) != A.shape[1]):
        print("Matrix times its tranpoise is not full rank")
    lhs= np.matmul(A.T,A)
    rhs= np.matmul(A.T,b)
    x=np.matmul(np.linalg.inv(lhs),rhs)
    return x
```

In [3]:
```
x, y, coefficients = make_regression(
    n_samples=50,
    n_features=1,
    n_informative=1,
    n_targets=1,
    noise=25,
    coef=True,
    random_state=1
)
```

We will now create an error function that will help us calculate the error between our predicted labels, and the true labels

Note on Mean sqared error (MSE)

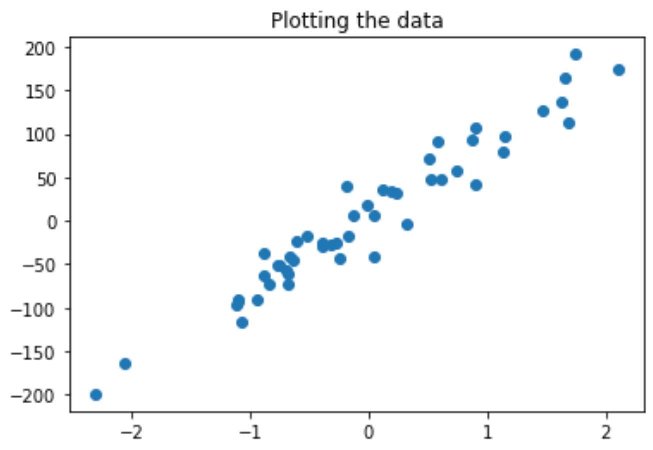
The MSE is a measure of the quality of an estimator. It calculates the mean squared difference across two quantities. The mean squared error is never negative, and the closer to zero, the better the estimate

In [4]:
```
def MSE(y_true,y_predicted):
    return (1/len(y_true))*np.sum((np.subtract(y_true,y_predicted))**2)
```

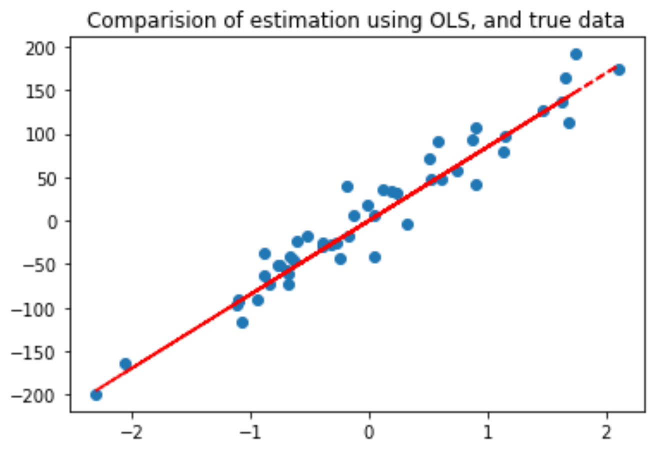
To begin with, let's visualize our data, I purposely picked a noisy set of labels, to show the difference in OLS with noise in the labels

```
In [5]: plt.scatter(x,y)
        plt.title("Plotting the data")
        plt.show
```

```
Out[5]: <function matplotlib.pyplot.show(*args, **kw)>
```



```
In [6]: weights_predicted= OLS(x,y)
        y_predicted= np.matmul(x,weights_predicted)
        error= MSE(y,y_predicted)
        plt.scatter(x,y)
        plt.plot(x,y_predicted,linestyle='dashed',c='red')
        plt.title("Comparision of estimation using OLS, and true data")
        plt.show()
```



```
In [7]: print("The mean squared error of the predicted labels using OLS is " + str(erro
        r))
```

```
The mean squared error of the predicted labels using OLS is 444.48185006939656
```

1.2 Ridge Regression

When the numerical features of the data in question are collinear, there arises a few issues of numerical instability and generalization. Ordinary least squares does not handle this well and in order to mitigate this we add a regularization term that penalizes data points that may represent outliers in the dataset

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
from sklearn.datasets import make_regression
```

In [2]:
```
#This regression method is more apt for when a certain data point is an anomaly
and may skew the results
#we add a reguralization paramater that helps control that
import numpy as np
def RidgeRegression(A,b,lambda_):
    #lambda_ here is the reguralization parameter,also solving Ax=b
    n, m = A.shape
    I = np.identity(m)
    x= np.dot(np.dot(np.linalg.inv(np.dot(A.T, A) + lambda_ * I), A.T), b)
    return x
```

Creating the data set below

In [3]:
```
X, y, coefficients = make_regression(
    n_samples=50,
    n_features=1,
    n_informative=1,
    n_targets=1,
    noise=5,
    coef=True,
    random_state=1
)
```

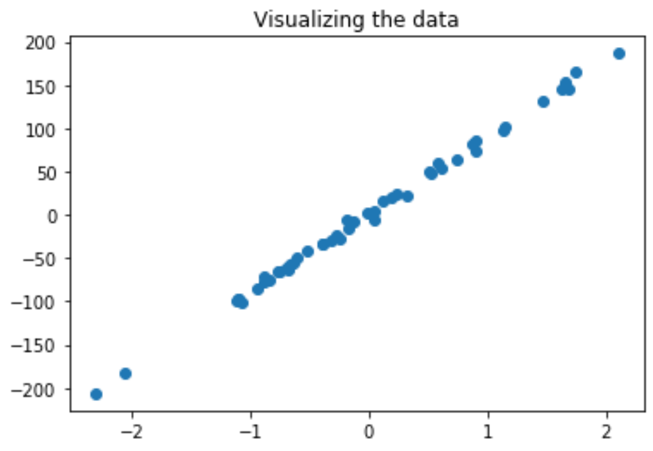
Note on Mean sqared error (MSE)

The MSE is a measure of the quality of an estimator. It calculates the mean squared difference across two quantities. The mean squared error is never negative, and the closer to zero, the better the estimate

In [4]:
```
def MSE(y_true,y_predicted):
    return (1/len(y_true))*np.sum((np.subtract(y_true,y_predicted))**2)
```

Let us first visualize the data, the noise is not very high in this sample

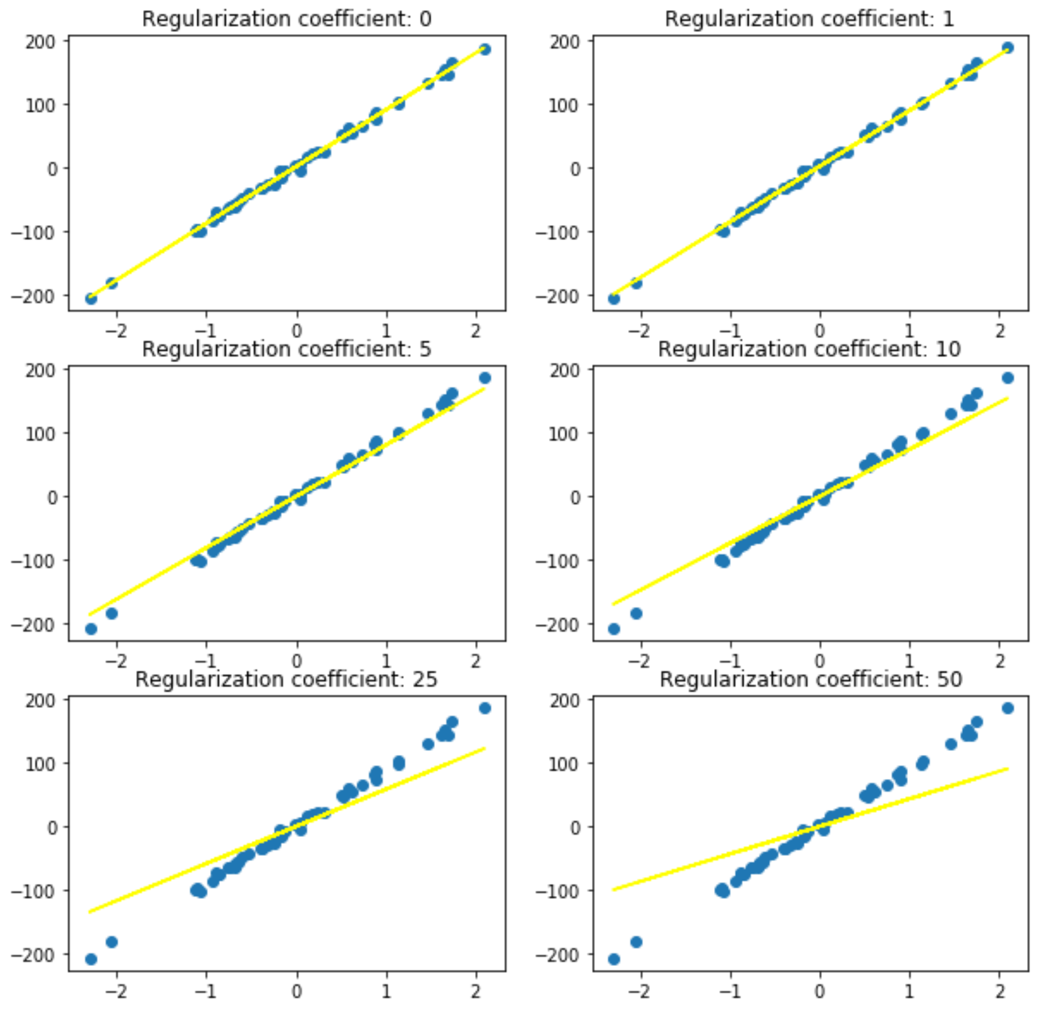
In [5]:
```
plt.scatter(X, y)
plt.title("Visualizing the data")
plt.show()
```



Now let us plot our estimate using various penalty coefficients

In [6]:
```
lambdas= [0,1,5,10,25,50]
plt.figure(figsize=(10,10))

for i in range(len(lambdas)):
    w= RidgeRegression(X,y,lambdas[i])
    pred_y= np.matmul(X,w)
    plt.subplot(3,2,i+1)
    plt.title("Regularization coefficient: " + str(lambdas[i]))
    plt.scatter(X, y)
    plt.plot(X, pred_y, c='yellow')
    mse= MSE(y,pred_y)
    print("The mean squared error using the penalty coefficient " + str(lambdas
[i]) + " is " + str(mse) )
plt.show()
```

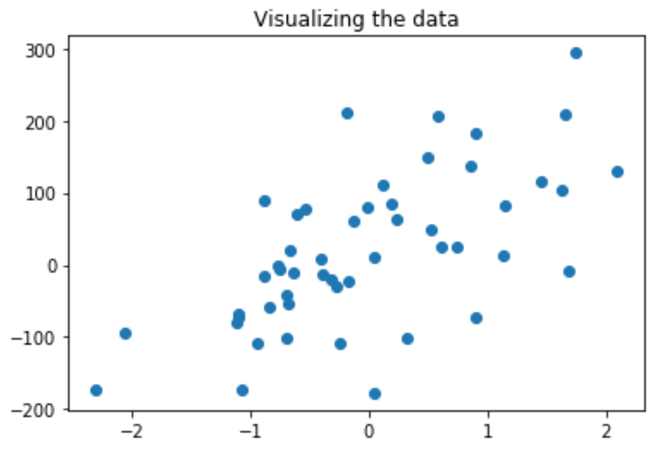
```
The mean squared error using the penalty coefficient 0 is 17.779274002775864
The mean squared error using the penalty coefficient 1 is 21.025088343365702
The mean squared error using the penalty coefficient 5 is 86.92923432921926
The mean squared error using the penalty coefficient 10 is 248.0104499012563
The mean squared error using the penalty coefficient 25 is 919.8665198890254
The mean squared error using the penalty coefficient 50 is 2006.3785724573142
```



So what's going on here, since the noise in the data set isn't very high, the regularization coefficent seems to being doing very little as it increases, at in unecessarily penalizes points that should't be penalized, but now, let us really increase the noise in the data set through outliers and observes what occurs.

In [7]:
```
x, Y, coefficients = make_regression(
    n_samples=50,
    n_features=1,
    n_informative=1,
    n_targets=1,
    noise=100,
    coef=True,
    random_state=1
)
```

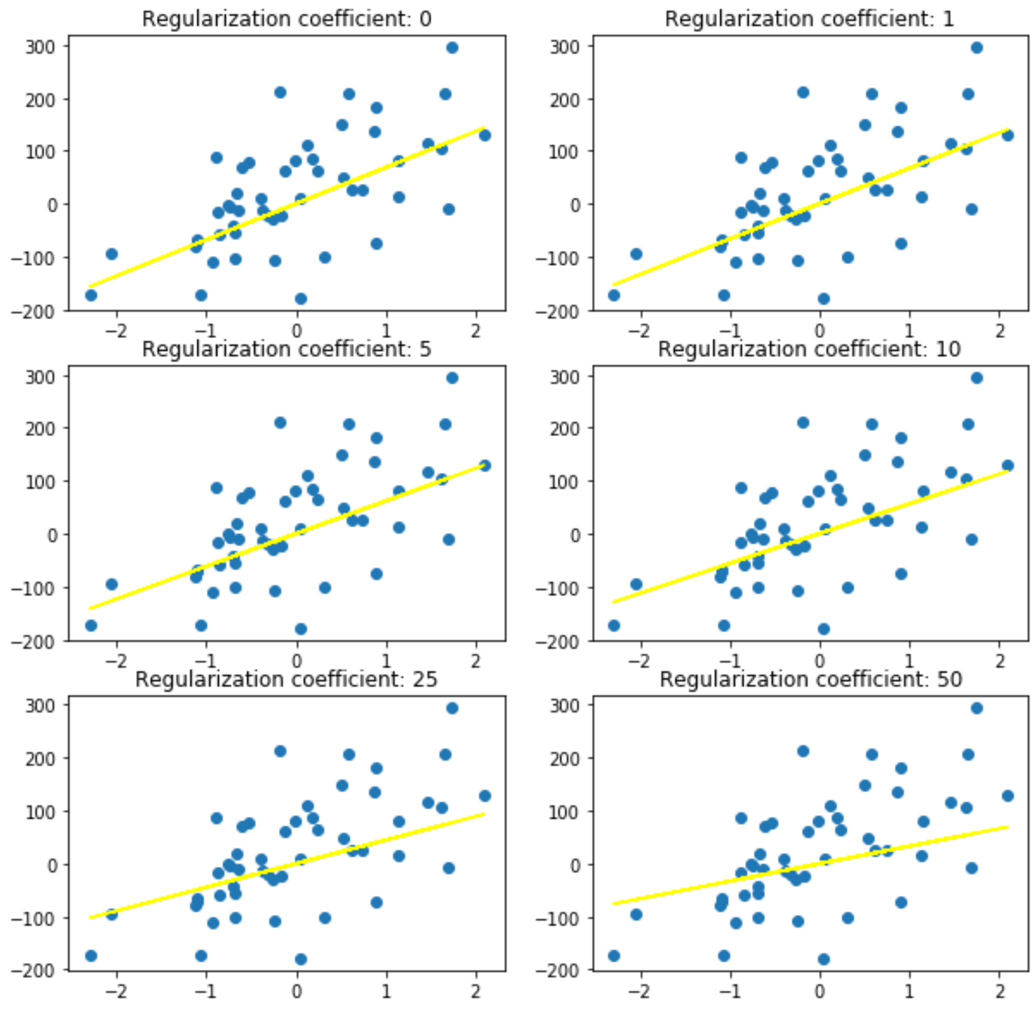
In [8]:
```
plt.scatter(x, Y)
plt.title("Visualizing the data")
plt.show()
```



As we can now see the data set is much noiser, let us see how our reguralization paramters perform now.

```
In [9]: lambdas= [0,1,5,10,25,50]
        plt.figure(figsize=(10,10))

        for i in range(len(lambdas)):
            w= RidgeRegression(x,Y,lambdas[i])
            pred_y= np.matmul(x,w)
            plt.subplot(3,2,i+1)
            plt.title("Regularization coefficient: " + str(lambdas[i]))
            plt.scatter(x, Y)
            plt.plot(x, pred_y, c='yellow')
            mse= MSE(Y,pred_y)
            print("The mean squared error using the penalty coefficient " + str(lambdas
        [i]) + " is " + str(mse) )
        plt.show()
```

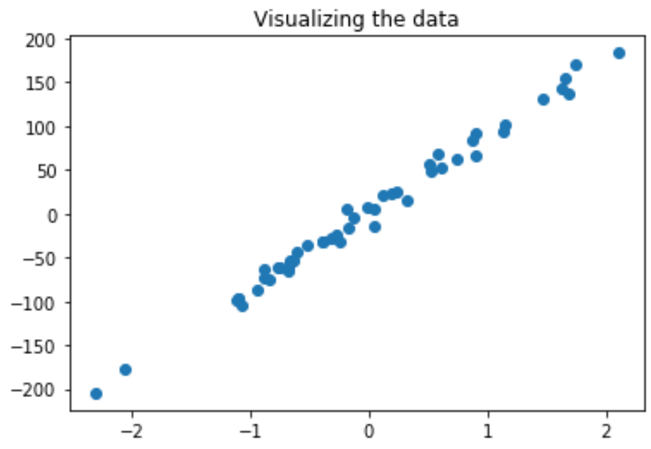
```
The mean squared error using the penalty coefficient 0 is 7111.709601110346
The mean squared error using the penalty coefficient 1 is 7113.601151011548
The mean squared error using the penalty coefficient 5 is 7152.007839671917
The mean squared error using the penalty coefficient 10 is 7245.880478531952
The mean squared error using the penalty coefficient 25 is 7637.415282884596
The mean squared error using the penalty coefficient 50 is 8270.59745229904
```



As we can know see the regularization parameters offer similar peformance in the case of increased noise, however, a highere reguralization paramter is not necessarily better, let us look at one last data set, with relatively homogenous data, with a few outliers

In [10]:
```
X, Y, coefficients = make_regression(
    n_samples=50,
    n_features=1,
    n_informative=1,
    n_targets=1,
    noise=10,
    coef=True,
    random_state=1
)
```

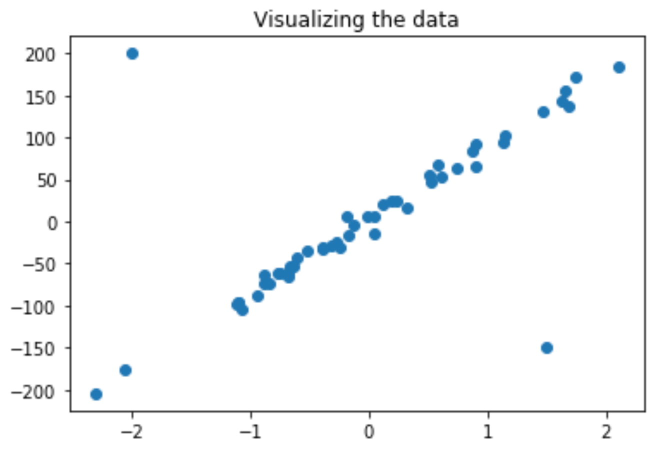
In [11]:
```
plt.scatter(X, Y)
plt.title("Visualizing the data")
plt.show()
```



In [12]:
```
Y= np.append(Y,[200,-150])
X= np.append(X,[-2,1.5])
X= np.expand_dims(X,axis=1)
print(np.shape(X))
print(np.shape(Y))
```

```
(52, 1)
(52,)
```

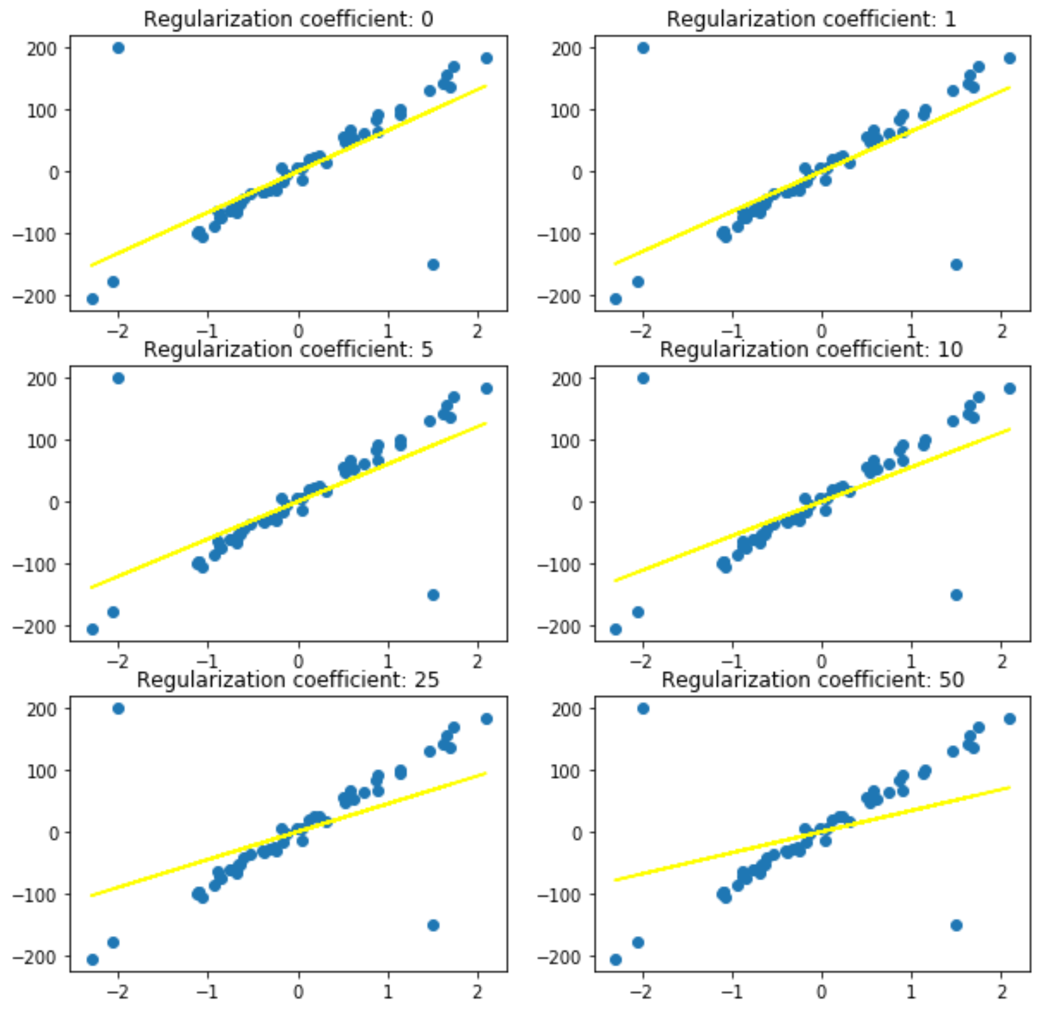
```
In [13]: plt.scatter(X, Y)
         plt.title("Visualizing the data")
         plt.show()
```



So now, we have some definitive outliers, let us us see what our regularization coefficients do now:

In [14]:
```
lambdas= [0,1,5,10,25,50]
plt.figure(figsize=(10,10))


for i in range(len(lambdas)):
    w= RidgeRegression(X,Y,lambdas[i])
    pred_y= np.matmul(X,w)
    plt.subplot(3,2,i+1)
    plt.title("Regularization coefficient: " + str(lambdas[i]))
    plt.scatter(X, Y)
    plt.plot(X, pred_y, c='yellow')
    mse= MSE(Y,pred_y)
    print("The mean squared error using the penalty coefficient " + str(lambdas
[i]) + " is " + str(mse) )

plt.show()
```

```
The mean squared error using the penalty coefficient 0 is 3822.9132381893432
The mean squared error using the penalty coefficient 1 is 3824.4303500563256
The mean squared error using the penalty coefficient 5 is 3855.8140551973756
The mean squared error using the penalty coefficient 10 is 3934.5434909947685
The mean squared error using the penalty coefficient 25 is 4278.859367887072
The mean squared error using the penalty coefficient 50 is 4870.677139425841
```



1.3 Weighted Least Squares

The idea of weighted least squares is that inherently, we care about some data points more than others, so we add a weighted matrix to the Ordinary least squares objective. The weighted least squares method can also be thought of as ordinary least squares with scaled feature and label data

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
from numpy.linalg import matrix_rank
%matplotlib inline
from sklearn.datasets import make_regression
```

In [2]:
```
#weighted least squares solves for the same Ax=b problem, but with a weight term
w on either side
import numpy as np
def WLS(A,b,w):
    #this code can be solved using a weighted A and b and just putting it throug
h regular OLS, but for the sake of clarity, I
    #will be writing it out in full
    weighted_A= np.matmul(np.sqrt(w),A)
    weighted_b= np.matmul(np.sqrt(w),b)
    lhs= np.matmul(weighted_A.T,weighted_A)
    rhs= np.matmul(weighted_A.T,weighted_b)
    x= np.matmul(np.linalg.inv(lhs),rhs)
    return x
#as we can observe from above, this is simply a OLS solution with weights on the
A and b components, determine from a weight
#vector w, thatr can be used to amnipulate the data
```

In [3]:
```
#this method is an ordinary least square solvers, good for datasets with uniform
data, no need for reguralization
def OLS(A,b):
    #solving the equation Ax=b
    if (matrix_rank(np.matmul(A.T,A)) != A.shape[1]):
        print("Matrix times its tranpoise is not full rank")
    lhs= np.matmul(A.T,A)
    rhs= np.matmul(A.T,b)
    x=np.matmul(np.linalg.inv(lhs),rhs)
    return x
```

In [4]:
```
x, y, coefficients = make_regression(
    n_samples=50,
    n_features=1,
    n_informative=1,
    n_targets=1,
    noise=25,
    coef=True,
    random_state=1
)
```

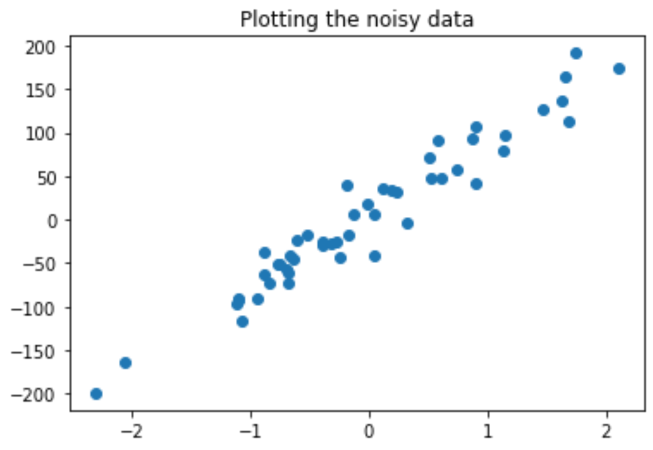
Note on Mean sqared error (MSE)

The MSE is a measure of the quality of an estimator. It calculates the mean squared difference across two quantities. The mean squared error is never negative, and the closer to zero, the better the estimate

In [5]:
```
def MSE(y_true,y_predicted):
    return (1/len(y_true))*np.sum((np.subtract(y_true,y_predicted))**2)
```

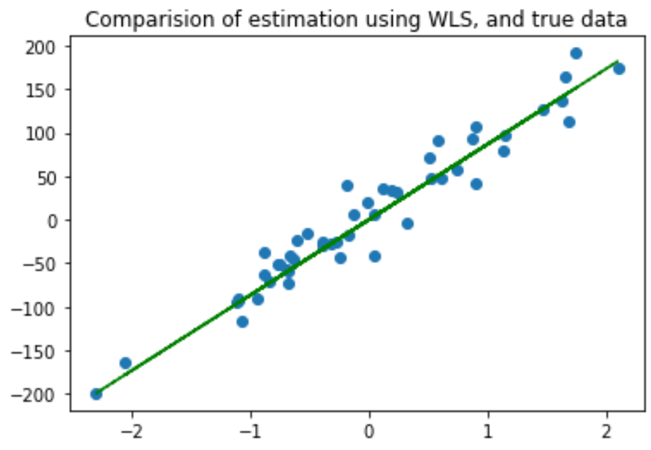
Visualizing the data

In [6]:
```
plt.scatter(x,y)
plt.title("Plotting the noisy data")
plt.show()
```



In [7]:
```
n= len(y)
vec=np.random.choice(np.arange(0,50,5),n)
w=np.diag(vec)
```

```
In [8]: weights_predicted= WLS(x,y,w)
        y_predicted= np.matmul(x,weights_predicted)
        error= MSE(y,y_predicted)
        plt.scatter(x,y)
        plt.plot(x,y_predicted,linestyle='dashed',c='green')
        plt.title("Comparision of estimation using WLS, and true data")
        plt.show()
```

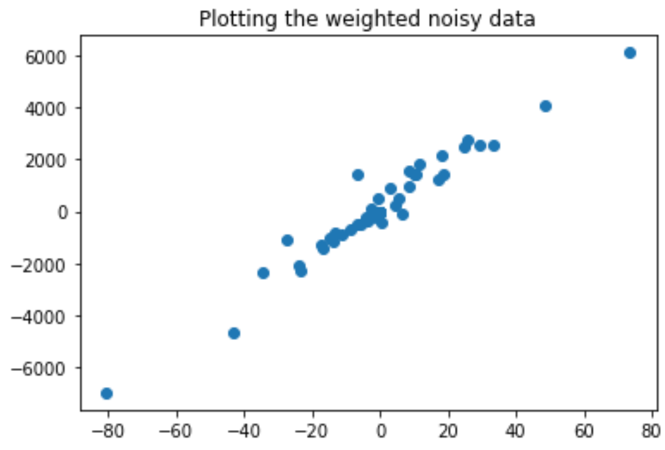


```
In [9]: print("The mean squared error of the predicted labels using OLS is " + str(erro
        r))
```

```
The mean squared error of the predicted labels using OLS is 447.79985496641115
```

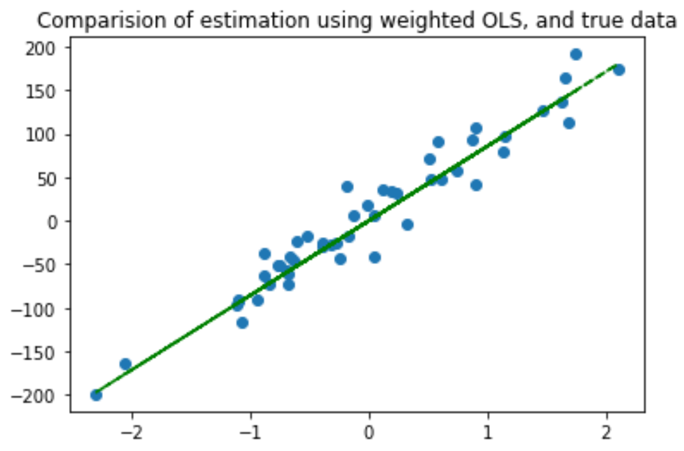
Note that the value above is different that what we saw with our OLS case, however, to prove a claim we made earlier, we are going to weight the data, then perform OLS and see what happens

```
In [10]: w_x= np.matmul(w,x)
         w_y= np.matmul(w,y)
         plt.scatter(w_x,w_y)
         plt.title("Plotting the weighted noisy data")
         plt.show()
```



Notice that the weighted noisy data looks very different, as it should.

```
In [11]: weights_predicted= OLS(w_x,w_y)
         y_predicted= np.matmul(x,weights_predicted)
         error= MSE(y,y_predicted)
         plt.scatter(x,y)
         plt.plot(x,y_predicted,linestyle='dashed', c='green')
         plt.title("Comparision of estimation using weighted OLS, and true data")
         plt.show()
```



```
In [12]: print("The mean squared error of the predicted labels using OLS is " + str(erro
         r))
```

```
The mean squared error of the predicted labels using OLS is 445.1414929065646
```

Notice that our error above is much closer to the WLS case, this is because in the case of performing OLS upon a weighted data set, you are simply performing WLS upon the original set (Note: the small difference in MSE is probably due to cutoffs at the sqrt fuction in the helper above)

1.4 Total Least Squares

Total Least Squares is considered an "errors-in-variables" model as in its usage both the numerical features (dependent variables) and the labels (independent variables) are corrupted by noise. Total Least Squares is also sometimes referred to as orthogonal regression.

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
from numpy.linalg import matrix_rank
%matplotlib inline
from sklearn.datasets import make_regression
```

In [2]:
```
# Total least squares assumes corruption in all variables, i.e both in input and
output data
#a great way to solve for the solution is by manipulating the k+1th singluar val
ue of the SVD decomposition of [A,b]
import numpy as np
def TLS(A,b):
    svd_matrix= np.vstack((A.T,b)).T
    U,sigma,V_transpose= np.linalg.svd(svd_matrix)
    V= V_transpose.T
    split= A.shape[1]
    Vyy= V[split:,split:]
    Vxy= V[:split,split:]
    sol_tls= -Vxy/Vyy
#now we can also return the denoised A and b as well
    V_split= V[:,split:]
    A_error= np.matmul(svd_matrix,V_split)
    A_error= -np.matmul(A_error,V_split.T)
    A_e= A_error[:,:split]
    A_tls= A + A_e
    b_tls= np.matmul(A_tls,sol_tls)
    return sol_tls,A_tls,b_tls
```

In [3]:
```
#this method is an ordinary least square solvers, good for datasets with uniform
data, no need for reguralization
def OLS(A,b):
    #solving the equation Ax=b
    if (matrix_rank(np.matmul(A.T,A)) != A.shape[1]):
        print("Matrix times its tranpoise is not full rank")
    lhs= np.matmul(A.T,A)
    rhs= np.matmul(A.T,b)
    x=np.matmul(np.linalg.inv(lhs),rhs)
    return x
```

Creating sample data courtesy of https://towardsdatascience.com/total-least-squares-in-comparison-with-ols-and-odr-f050ffc1a86a (https://towardsdatascience.com/total-least-squares-in-comparison-with-ols-and-odr-f050ffc1a86a)

```
In [4]: x, y, coefficients = make_regression(
            n_samples=50,
            n_features=1,
            n_informative=1,
            n_targets=1,
            noise=5,
            coef=True,
            random_state=1
        )
```

We will now create an error function that will help us calculate the error between our predicted labels, and the true labels

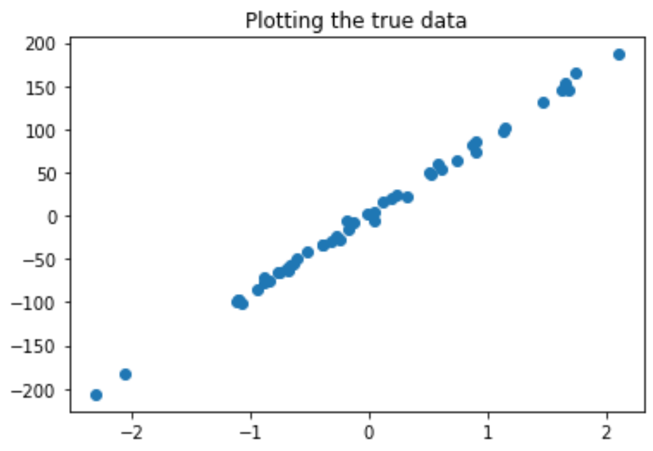
Note on Mean sqared error (MSE)

The MSE is a measure of the quality of an estimator. It calculates the mean squared difference across two quantities. The mean squared error is never negative, and the closer to zero, the better the estimate

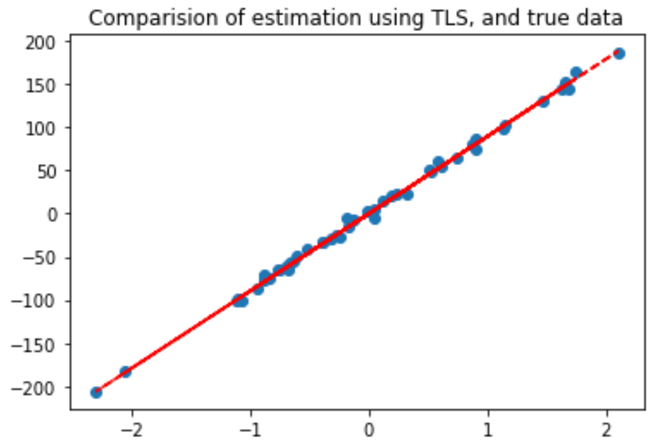
```
In [5]: def MSE(y_true,y_predicted):
            return (1/len(y_true))*np.sum((np.subtract(y_true,y_predicted))**2)
```

To begin with, let's visualize our data

```
In [6]: plt.scatter(x,y)
        plt.title("Plotting the true data")
        plt.show()
```



In [7]:
```
weights_predicted,A,b= TLS(x,y)
y_predicted= np.matmul(x,weights_predicted)
error= MSE(y,y_predicted)
plt.scatter(x,y)
plt.plot(x,y_predicted,linestyle='dashed',c='red')
plt.title("Comparision of estimation using TLS, and true data")
plt.show()
mse= MSE(y,y_predicted)
print("The mean squared error is " + str(mse) )
```

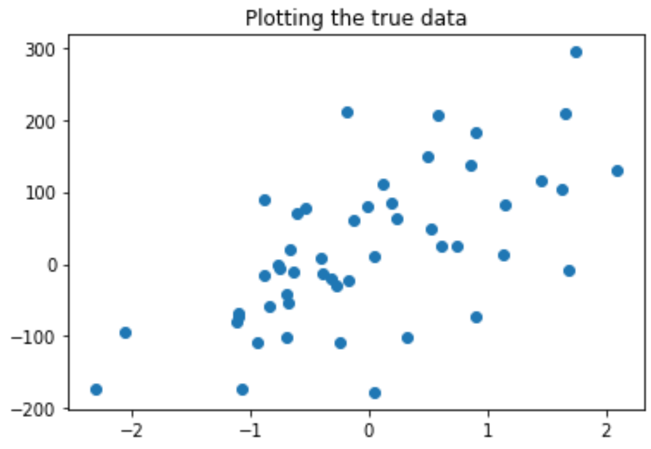


```
The mean squared error is 751409.7183623331
```

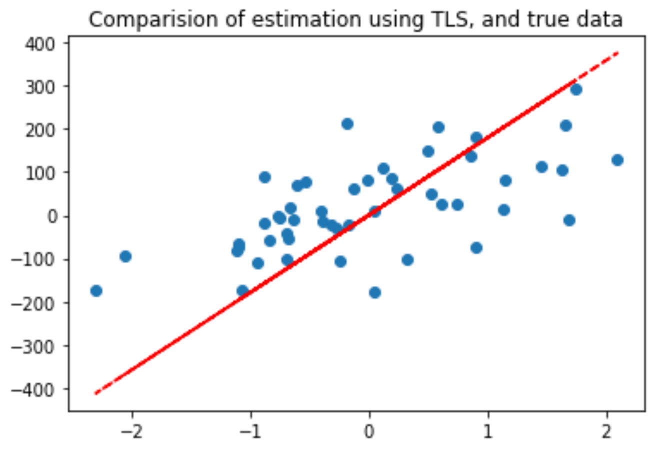
Let us up the noise and see what happens

In [9]:
```
x, y, coefficients = make_regression(
    n_samples=50,
    n_features=1,
    n_informative=1,
    n_targets=1,
    noise=100,
    coef=True,
    random_state=1
)
```

In [10]:
```
plt.scatter(x,y)
plt.title("Plotting the true data")
plt.show()
```

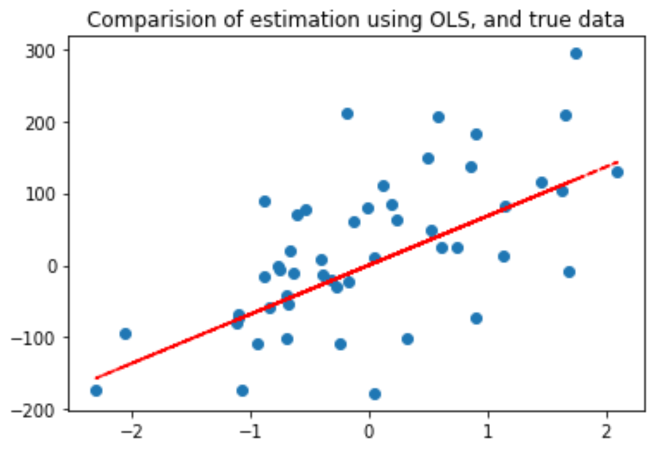


In [11]:
```
weights_predicted,A,b= TLS(x,y)
y_predicted= np.matmul(x,weights_predicted)
error= MSE(y,y_predicted)
plt.scatter(x,y)
plt.plot(x,y_predicted,linestyle='dashed',c='red')
plt.title("Comparision of estimation using TLS, and true data")
plt.show()
```



Comparing to ordinary least squares:

In [12]:
```
weights_predicted= OLS(x,y)
y_predicted= np.matmul(x,weights_predicted)
error= MSE(y,y_predicted)
plt.scatter(x,y)
plt.plot(x,y_predicted,linestyle='dashed',c='red')
plt.title("Comparision of estimation using OLS, and true data")
plt.show()
```



2 Correlation

2.1 Principal Component Analysis

Input: a matrix X, unlabeled data matrix

Output: k directions of maximum variation

PCA is a data feature reduction technique that can be especially useful in machine learning applications. As a general data analysis tool, PCA yields directions of maximum variance of the inputted data, and the corresponding amount of variance captured by each of those directions. We choose some k, a hyperparameter, which determines the number of vectors or 'directions' PCA returns.

As a dimensionality reduction tool, often when we work with data in machine learning, the number of features (variables) can often be too numerous to work with easily. The general issue is of increased computational complexity. Instead, we can use PCA to choose the k most varied directions in the inputted data, and then project the input data matrix onto those directions, resulting in only k features.

Note on hyperparameters: hyperparameters, such as k in PCA, are not values determined by the algorithm to be the best for the problem. As such we must think of ways to effectively validate our choice for k and compare how different hyperparameter values effect our model. Here for example, we would k to capture a large portion of the variance.

In [0]:
```
import numpy as np
```

In [0]:
```
def scratch_PCA(X, k):
    demeaned_X= X - np.mean(X, axis=0)
    #we now have to caculate the covariance matrix
    cov_X= np.matmul(np.transpose(demeaned_X), demeaned_X)
    sigma,V= np.linalg.eig(cov_X)
    # you then select the k biggest eigenvalues and their corresponding eigenvec
tors
    return sigma[:k], V[:k]
```

In [0]:
```
def svd_PCA(X,k):
  U, S, V = np.linalg.svd(X - np.mean(X, axis=0))
  return S[:k]**2, V[:k]
```

Explanation

In PCA we choose the first k components of the SVD of a matrix. In a nutshell, SVD represents the breakdown of the input into a sum of rank 1 matrices. By ordering the variance (the sigma values) by high to low, we capture the largest amount of variance in the first k components. As we increase k to the rank of the input, we fully capture the data matrix.

Mathematically:

SVD is represented as U *S* V.T

- U - eigenvectors of X @ X.T
- V - eigenvectors of X.T @ X
- S - sigma squared = eigenvalues of both

PCA uses the first k values of S and the first k vectors of V

```
In [0]: from sklearn.decomposition import PCA
```

In practice we would use modules such as sci-kit learn for running methods such as PCA.

Example extrapolated method below:

```
In [0]: def sk_PCA(X, k):
          # initialize pca with num of components (k)
          pca = PCA(n_components=k)

          # run pca on our data set
          pca.fit(X)

          # get our first k components from our solutions
          variance = pca.explained_variance_[:k]
          vectors = pca.components_[:k]
          vectors = [np.asarray(vectors[i]) for i in range(len(vectors))]

          return variance, vectors
```

2.1.1 Examples

Let's start with our implementations.

```
In [0]: import numpy as np
        import pandas as pd
        from sklearn.decomposition import PCA
        from sklearn.preprocessing import StandardScaler
        import matplotlib.pyplot as plt
```

```
In [0]: def scratch_PCA(X, k):
          demeaned_X= X - np.mean(X, axis=0)
          #we now have to caculate the covariance matrix
          cov_X= np.matmul(np.transpose(demeaned_X), demeaned_X)
          sigma,V= np.linalg.eig(cov_X)
          # you then select the k biggest eignevalues and their corresponding eigenvecto
        rs
          return sigma[:k], V[:k]
```

```
In [0]: def sklearn_PCA(X, k):
          # initialize pca with num of components (k)
          pca = PCA(n_components=k)

          # run pca on our data set
          pca.fit(X)

          # get our first k components from our solutions
          variance = pca.explained_variance_[:k]
          vectors = pca.components_[:k]
          vectors = [np.asarray(vectors[i]) for i in range(len(vectors))]

          return variance, vectors
```

We use a standard dataset (Iris) included with scikit-learn

```
In [0]: url = "https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data"
        features = ['sepal length', 'sepal width', 'petal length', 'petal width']
        iris_df = pd.read_csv(url, names=['sepal length','sepal width','petal length',
                                          'petal width','target'])

        iris_df.head()
```

Out[0]:

	sepal length	sepal width	petal length	petal width	target
0	5.1	3.5	1.4	0.2	Iris-setosa
1	4.9	3.0	1.4	0.2	Iris-setosa
2	4.7	3.2	1.3	0.2	Iris-setosa
3	4.6	3.1	1.5	0.2	Iris-setosa
4	5.0	3.6	1.4	0.2	Iris-setosa

```
In [0]: # extract our data and labels
        X = iris_df.loc[:, features].values
        y = iris_df.loc[:,['target']].values
        X = StandardScaler().fit_transform(X)
```

The dataset (iris) comes with several *features* and a target, which corresponds to the classification for the row with the given values for each feature.

For PCA we just look at the features to determine the directions of maximum variance, and then we can use our principal components to map back to the targets.

Another important note: we use standard scaler on our input X as PCA is not scale invariant. In other words, by scaling all values by the same factor, our results may change. Example as follows:

Effect of Scale on PCA:

```
In [0]: scale_ex = np.array([[1,-5],[-3,-8],[7, -3]])
        scale_ex

Out[0]: array([[ 1, -5],
               [-3, -8],
               [ 7, -3]])
```

```
In [0]: sklearn_PCA(scale_ex, 2)

Out[0]: (array([31.40128474,  0.26538192]),
         [array([0.89728145, 0.44145894]), array([-0.44145894,  0.89728145])])
```

```
In [0]: sklearn_PCA(2*scale_ex, 2)

Out[0]: (array([125.60513897,   1.06152769]),
         [array([0.89728145, 0.44145894]), array([-0.44145894,  0.89728145])])
```

As we see above, the amount of variance captured by the first principal component is not scaled by 2, but by the scalar squared.

Back to the Iris dataset:

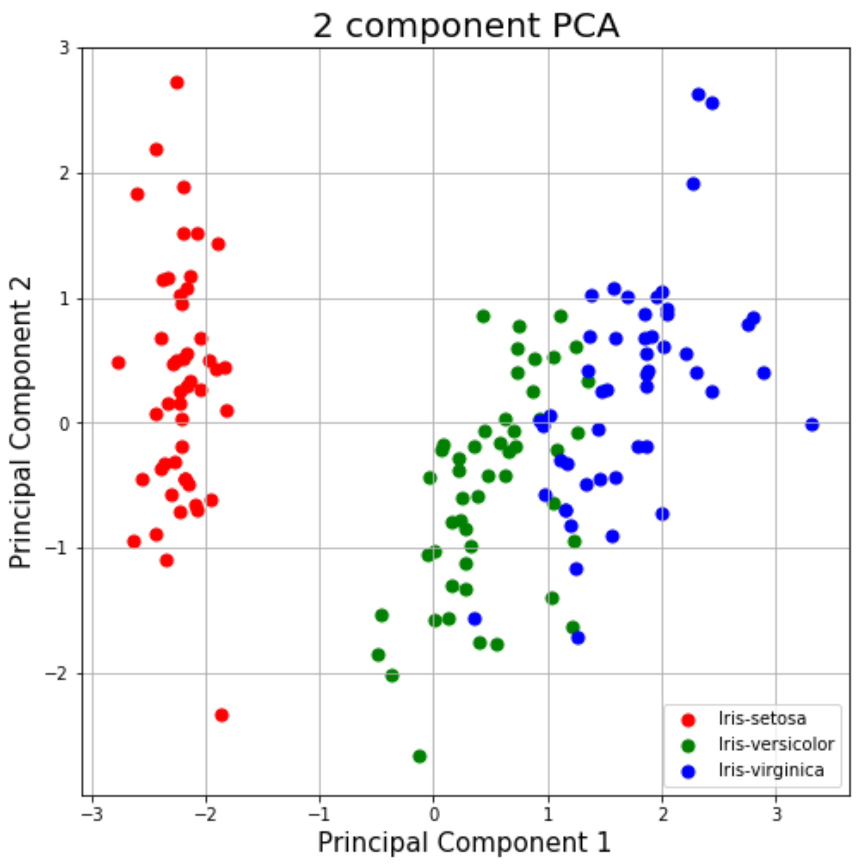
After we obtain the principal components, we project the original data using the principal component vectors.

This projection demonstrates why PCA is dimension reductionality:

```
In [0]: variance, pc = sklearn_PCA(X, 2)
        projected = np.matmul(X, np.transpose(np.asarray(pc)))
        principalDf = pd.DataFrame(data = projected
                     , columns = ['principal component 1', 'principal component 2'])
        # add back the target
        finalDf = pd.concat([principalDf, iris_df[['target']]], axis = 1)
```

Visualization

In [0]:
```python
fig = plt.figure(figsize = (8,8))
ax = fig.add_subplot(1,1,1)
ax.set_xlabel('Principal Component 1', fontsize = 15)
ax.set_ylabel('Principal Component 2', fontsize = 15)
ax.set_title('2 component PCA', fontsize = 20)
targets = ['Iris-setosa', 'Iris-versicolor', 'Iris-virginica']
colors = ['r', 'g', 'b']
for target, color in zip(targets,colors):
    indicesToKeep = finalDf['target'] == target
    ax.scatter(finalDf.loc[indicesToKeep, 'principal component 1']
               , finalDf.loc[indicesToKeep, 'principal component 2']
               , c = color
               , s = 50)
ax.legend(targets)
ax.grid()
```



In [0]:
```python
print("Variance explained by Principal Component 1: " + str(variance[0]))
print("Variance explained by Principal Component 2: " + str(variance[1]))
```

```
Variance explained by Principal Component 1: 2.930353775589314
Variance explained by Principal Component 2: 0.9274036215173412
```

As we see above, by using the first 2 principal components, we were able to compress our dataset with 4 initial variables/features, into 2 features and visualize it on a simple plot. We also see how variance factors into the split of the principal components. The first principal component shows the best split between the data, and very clearly sets apart the two plant species, despite the PCA being carried out on data without any labels (knowledge of which species a row belongs to). Accordingly, the first principal component also captures the most variance of the original dataset.

Credit:

Visualization: https://towardsdatascience.com/pca-using-python-scikit-learn-e653f8989e60 (https://towardsdatascience.com/pca-using-python-scikit-learn-e653f8989e60)

2.2 Canonical Correlation Analysis

In PCA we looked at only the input X to carry out dimensionality reduction and is unsupervised in that sense. CCA adds invariance and incorporates the labels y into the analysis.

In [0]:
```
import numpy as np
from sklearn.cross_decomposition import CCA
```

In [0]:
```
# some dummy data for showing the process of running CCA
X = np.array([[0., 0., 1.], [1.,0.,0.], [2.,2.,2.], [3.,5.,4.]])
y = np.array([[0.1, -0.2], [0.9, 1.1], [6.2, 5.9], [11.9, 12.3]])
print(X)
print(y)
```

```
[[0. 0. 1.]
 [1. 0. 0.]
 [2. 2. 2.]
 [3. 5. 4.]]
[[ 0.1 -0.2]
 [ 0.9  1.1]
 [ 6.2  5.9]
 [11.9 12.3]]
```

In [0]:
```
cca = CCA(n_components=1)
cca.fit(X, y)
```

Out[0]:
```
CCA(copy=True, max_iter=500, n_components=1, scale=True, tol=1e-06)
```

In [0]:
```
# more useful than simply the fit, the actual transformation from our computed C
CA
X_c, y_c = cca.fit_transform(X, y)
print(X_c)
print(y_c)
```

```
[[-1.3373174 ]
 [-1.10847164]
 [ 0.40763151]
 [ 2.03815753]]
[[-0.85511537]
 [-0.70878547]
 [ 0.26065014]
 [ 1.3032507 ]]
```

To demonstrate why CCA is more powerful, let's examine scaling the data.

```
In [0]: X_c, y_c = cca.fit_transform(2*X, 2*y)
        print(X_c)
        print(y_c)
```

```
[[-1.3373174 ]
 [-1.10847164]
 [ 0.40763151]
 [ 2.03815753]]
[[-0.85511537]
 [-0.70878547]
 [ 0.26065014]
 [ 1.3032507 ]]
```

As we see, scaling both by 2 doesn't change the transform at all. Going further:

```
In [0]: X_c, y_c = cca.fit_transform(2*X, 3*y)
        print(X_c)
        print(y_c)
```

```
[[-1.3373174 ]
 [-1.10847164]
 [ 0.40763151]
 [ 2.03815753]]
[[-0.85511537]
 [-0.70878547]
 [ 0.26065014]
 [ 1.3032507 ]]
```

Even scaling separately, the transformation remains the same. This shows a very important porperty of CCA - it's invariant to affine transformations (scaling or addition).

Proof:

```
In [0]: from IPython.display import Image
        Image(filename="/content/data/screenshot.PNG")
```

Out[0]:

$$\begin{aligned}\rho(aX+b, cY+d) &= \frac{\operatorname{Cov}(aX+b, cY+d)}{\sqrt{\operatorname{Var}(aX+b)\operatorname{Var}(cY+d)}}\\ &= \frac{\operatorname{Cov}(aX, cY)}{\sqrt{\operatorname{Var}(aX)\operatorname{Var}(cY)}}\\ &= \frac{a\cdot c\cdot\operatorname{Cov}(X, Y)}{\sqrt{a^2\operatorname{Var}(X)\cdot c^2\operatorname{Var}(Y)}}\\ &= \frac{\operatorname{Cov}(X, Y)}{\sqrt{\operatorname{Var}(X)\operatorname{Var}(Y)}}\\ &= \rho(X, Y)\end{aligned}$$

Proof Source: EECS 189 Course Notes

2.3 PCA vs CCA - Multilabel Classification

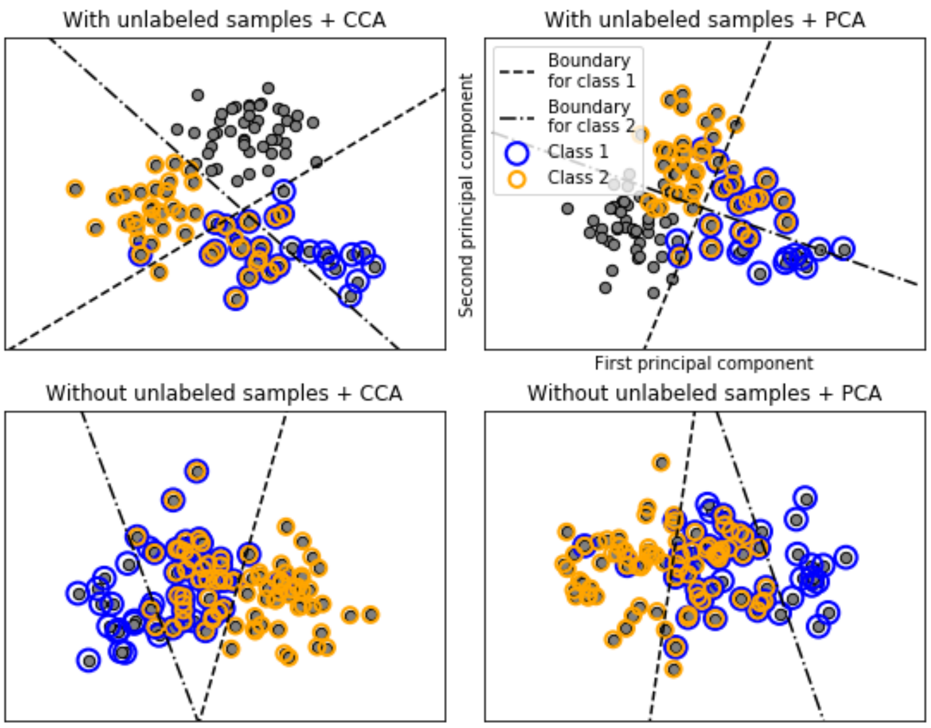
Here we look at an example comparing the classification of some data after projection using PCA and CCA techniques.

In [0]:
```
import numpy as np
import matplotlib.pyplot as plt

from sklearn.datasets import make_multilabel_classification
from sklearn.multiclass import OneVsRestClassifier
from sklearn.svm import SVC
from sklearn.decomposition import PCA
from sklearn.cross_decomposition import CCA
```

In [0]:
```
def plot_hyperplane(clf, min_x, max_x, linestyle, label):
    # get the separating hyperplane
    w = clf.coef_[0]
    a = -w[0] / w[1]
    xx = np.linspace(min_x - 5, max_x + 5)  # make sure the line is long enough
    yy = a * xx - (clf.intercept_[0]) / w[1]
    plt.plot(xx, yy, linestyle, label=label)
```

```
In [0]: def plot_subfigure(X, Y, subplot, title, transform):
            if transform == "pca":
                X = PCA(n_components=2).fit_transform(X)
            elif transform == "cca":
                X = CCA(n_components=2).fit(X, Y).transform(X)
            else:
                raise ValueError

            min_x = np.min(X[:, 0])
            max_x = np.max(X[:, 0])

            min_y = np.min(X[:, 1])
            max_y = np.max(X[:, 1])

            classif = OneVsRestClassifier(SVC(kernel='linear'))
            classif.fit(X, Y)

            plt.subplot(2, 2, subplot)
            plt.title(title)

            zero_class = np.where(Y[:, 0])
            one_class = np.where(Y[:, 1])
            plt.scatter(X[:, 0], X[:, 1], s=40, c='gray', edgecolors=(0, 0, 0))
            plt.scatter(X[zero_class, 0], X[zero_class, 1], s=160, edgecolors='b',
                        facecolors='none', linewidths=2, label='Class 1')
            plt.scatter(X[one_class, 0], X[one_class, 1], s=80, edgecolors='orange',
                        facecolors='none', linewidths=2, label='Class 2')

            plot_hyperplane(classif.estimators_[0], min_x, max_x, 'k--',
                            'Boundary\nfor class 1')
            plot_hyperplane(classif.estimators_[1], min_x, max_x, 'k-.',
                            'Boundary\nfor class 2')
            plt.xticks(())
            plt.yticks(())

            plt.xlim(min_x - .5 * max_x, max_x + .5 * max_x)
            plt.ylim(min_y - .5 * max_y, max_y + .5 * max_y)
            if subplot == 2:
                plt.xlabel('First principal component')
                plt.ylabel('Second principal component')
                plt.legend(loc="upper left")
```

```
In [4]: plt.figure(figsize=(8, 6))

        X, Y = make_multilabel_classification(n_classes=2, n_labels=1,
                                              allow_unlabeled=True,
                                              random_state=1)

        plot_subfigure(X, Y, 1, "With unlabeled samples + CCA", "cca")
        plot_subfigure(X, Y, 2, "With unlabeled samples + PCA", "pca")

        X, Y = make_multilabel_classification(n_classes=2, n_labels=1,
                                              allow_unlabeled=False,
                                              random_state=1)

        plot_subfigure(X, Y, 3, "Without unlabeled samples + CCA", "cca")
        plot_subfigure(X, Y, 4, "Without unlabeled samples + PCA", "pca")

        plt.subplots_adjust(.04, .02, .97, .94, .09, .2)
        plt.show()
```



Source: scikit-learn multilabel classification example

3 Classification

3.1 K Nearest Neighbors

KNN is a classification ML algorithm that works with the following process:

1. Calculate the distance to every point
2. Choose the k closest points
3. Each point gets an equal vote for classifying the target point

KNN can also be used for regression

KNN is a simple example of a non-parametric model - a model which "memorizes" all of the training data and uses it at test time.

As such there is no training phase and all of training data is used, meaning that the method is very computational expensive especially as the amount of data increases.

Let's look at an implementation of KNN from scratch:

In [0]:
```
from collections import Counter
import math
import numpy as np
```

In [0]:
```
def mean(labels):
  return np.sum(labels) / np.size(labels)

def mode(labels):
  return Counter(labels).most_common(1)[0][0]

def eucl_dist(p1, p2):
  axis_val = 0
  if len(p1.shape) != 1:
    axis_val = 1
  return np.sqrt(np.sum(np.power(np.subtract(p1, p2), 2), axis = axis_val))
```

The above are helper methods to generalize the KNN algorithm that follows.

As we see above, we've defined a distance function (Euclidean used here) and 2 choice functions:

- Mean will average the labels of the k nearest neighbors providing a regression answer
- Mode will yield the most common label of the k nearest neighbors, classifying the query data point

In [0]:
```
def knn(data, query, k, distance_fn, choice_fn):

  data = np.asarray(data)

  #strip labels, assumed to be last column of data
  features = data[:,:-1]
  labels = data[:,-1:].flatten()

  distance = distance_fn(features, query)

  # if distance doesn't have the same length, distance_fn isn't properly vectori
zed
  assert(len(distance) == len(data))

  #sort features and labels by distance
  inds = distance.argsort()
  sort_feats = features[inds]
  sort_labels = labels[inds]

  #return k closest indices, points and label
  #choose label based on choice_fn, mode for classification, mean for regression
  return dict(zip(inds[:k], sort_feats[:k])), choice_fn(sort_labels[:k])
```

A simple example follows (credit: https://towardsdatascience.com/machine-learning-basics-with-the-k-nearest-neighbors-algorithm-6a6e71d01761 (https://towardsdatascience.com/machine-learning-basics-with-the-k-nearest-neighbors-algorithm-6a6e71d01761))

In [0]:
```
#data with 1 feature, and last column as labels, 0 or 1
clf_data = [
       [22, 1],
       [23, 1],
       [21, 1],
       [18, 1],
       [19, 1],
       [25, 0],
       [27, 0],
       [29, 0],
       [31, 0],
       [45, 0]]
```

We run KNN here with the eucledian distance function and as a classifier
(using mode instead of mean).

In [0]:
```
neigh, pred = knn(clf_data, [33], 3, eucl_dist, mode)
```

In [0]:
```
print("The k closest points from the dataset to our query point:")
print(np.asarray(list(neigh.values())).flatten())
```

```
The k closest points from the dataset to our query point:
[31 29 27]
```

In [0]:
```
print("Label classification for the query point:")
print(pred)
```

```
Label classification for the query point:
0
```

3.1.1 Examples

Here we'll look at some examples and visualization of using the K nearest neighbors algorithm.

Let's start with the Iris dataset again (which details some features of leaves and a label)

In [0]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

In [3]:
```
url = "https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data"

# Assign colum names to the dataset
names = ['sepal-length', 'sepal-width', 'petal-length', 'petal-width', 'Class']

# Read dataset to pandas dataframe
dataset = pd.read_csv(url, names=names)

dataset.head()
```

Out[3]:

	sepal-length	sepal-width	petal-length	petal-width	Class
0	5.1	3.5	1.4	0.2	Iris-setosa
1	4.9	3.0	1.4	0.2	Iris-setosa
2	4.7	3.2	1.3	0.2	Iris-setosa
3	4.6	3.1	1.5	0.2	Iris-setosa
4	5.0	3.6	1.4	0.2	Iris-setosa

Given this dataset, we need to do some basic data cleaning to easily apply KNN.

In [0]:
```
X = dataset.iloc[:, :-1].values
y = dataset.iloc[:,-1].values
```

A commonly applied practice in ML is to randomly split our data into training and testing sets. While KNN is a non-parametric model, this is still good practice (when we test our model, our test points won't already be in the model).

In [0]:
```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.20)
```

Another good practice, especially applicable to KNN, is to normalize our features. This factors into the evaluation of the Euclidean distances that will be a core component of the KNN algorithm - normalizing prevents features with larger ranges from overinfluencing the classification.

```
In [0]: from sklearn.preprocessing import StandardScaler
        scaler = StandardScaler()
        scaler.fit(X_train)

        X_train = scaler.transform(X_train)
        X_test = scaler.transform(X_test)
```

The KNN python notebook details an implentation from scratch, but as is the case often in practice, we'll use an out of the box method.

```
In [0]: from sklearn.neighbors import KNeighborsClassifier
        classifier = KNeighborsClassifier(n_neighbors=5)
        classifier.fit(X_train, y_train)
        y_pred = classifier.predict(X_test)
```

Now y_pred refers to the predicted labels from our KNN classifier with 5 neighbors carried out on the training set partition. We check the accuracy of this prediction by comparing to y_test.

```
In [8]: print("Accuracy of KNN classifier with 5 neighbors:")
        accuracy = np.count_nonzero(y_pred == y_test) / len(y_test)
        accuracy = int(accuracy * 10000) / 100
        print(accuracy, "% with", len(y_test), "test points")
```

```
Accuracy of KNN classifier with 5 neighbors:
86.66 % with 30 test points
```

Again we can easily use an out of the box method to check the results of our classifier.

```
In [9]: from sklearn.metrics import classification_report, confusion_matrix
        print(confusion_matrix(y_test, y_pred))
        print(classification_report(y_test, y_pred))
```

```
[[ 8  0  0]
 [ 0  7  1]
 [ 0  3 11]]
                 precision    recall  f1-score   support

    Iris-setosa       1.00      1.00      1.00         8
Iris-versicolor       0.70      0.88      0.78         8
 Iris-virginica       0.92      0.79      0.85        14

       accuracy                           0.87        30
      macro avg       0.87      0.89      0.87        30
   weighted avg       0.88      0.87      0.87        30
```

The confusion matrix essentially describes the false positives/negatives and correct classifications into the labels. The left axis refers to true label while the right axis refers to labels our classifier outputted for the test points.Thus we can interpret the confusion matrix and see that two of the *versicolor* labelled test points were confused for *virginica*.

Determining K

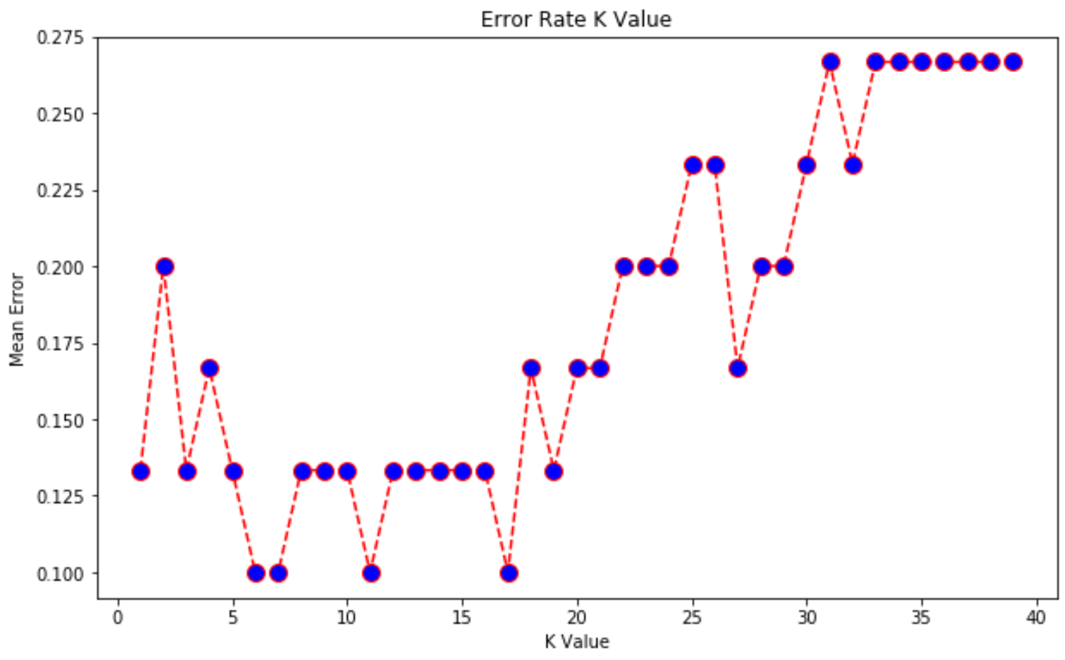
In the example above, we used k = 5 as the number of nearest neighbors to compare with. However, this k value is a **hyperparameter** - it isn't something determined by the algorithms but instead is something we set before the algorithm is run. It also has a very large influence on the accuracy of KNN.

Let's try some experiments and then interpret the results in terms of theory. In practice, the exact number used would be decided using a training and validation set, with the algorithm run on the training with different k values, and determining the best value for k based on scores on the validation set(s). Finally to evaluate the model it would be run on test set. (In the following we're effectively not carrying out the final testing step as it should never be used to determine the model / hyperparameters).

In [0]:
```
error = []

# Calculating error for K values between 1 and 40
for i in range(1, 40):
    knn = KNeighborsClassifier(n_neighbors=i)
    knn.fit(X_train, y_train)
    pred_i = knn.predict(X_test)
    error.append(np.mean(pred_i != y_test))
```

In [11]:
```
plt.figure(figsize=(10, 6))
plt.plot(range(1, 40), error, color='red', linestyle='dashed', marker='o',
         markerfacecolor='blue', markersize=10)
plt.title('Error Rate K Value')
plt.xlabel('K Value')
plt.ylabel('Mean Error')
```

Out[11]:
```
Text(0, 0.5, 'Mean Error')
```



As we see above, there are a few K values that result in a 0 error. This is because our dataset is very neatly organized and simple and KNN is a fairly powerful classifier despite being so straightforward. However, even here we see some generalities of how different values for k can influence our classification.

First, consider low k. Starting with k = 1, we know that if the test point is in the training set, it would "find" itself and thus always be completely accurate. If it doesn't exist, it would just look for the closest point and use its label.

While this could be very accurate (especially if the test point already was used when training the model), outliers would not be mitigated at all in this approach, Increasing k would allow our model to be less sensitive to outliers.

Consider:

- A region with several points with label 1
- In it, an outlier with label 0
- Increasing K forces a point placed in that region to look past that outlier and at the other K-1 nearest neighbors

However, if we keep increasing K and as we approach N (the number of points in the training data set), we end up overruling regions. At K = N, we assign every point the more frequent label in the training data set, which depending on the underlying distribution, could yield a very high error.

Such pitfalls can easily be avoided with a little bit of mindful selection and some validation to determine a good value for K.

Source: https://stackabuse.com/k-nearest-neighbors-algorithm-in-python-and-scikit-learn/ (https://stackabuse.com/k-nearest-neighbors-algorithm-in-python-and-scikit-learn/)

3.2 K Means

K means is a classification ML algorithm that works as follows:

1. Intialization of k centroids (randomly chosen)
2. Assigning each point in the data set to its closest cluster (via a distance parameter to the centroid such as the euclidean distance)
3. Moving the centroids of the clusters

In K means clustering, we keep iterating through the set until our centroids have no longer changed (i.e they have converged), and every point in the data set is assigned to a certain cluster.

K means is a discriminative method, that hard assigns data points to a given cluster.

In [1]:
```
import numpy as np
import math
from matplotlib import pyplot as plt
from copy import deepcopy
from sklearn.datasets.samples_generator import make_blobs
from sklearn.metrics import mean_squared_error
```

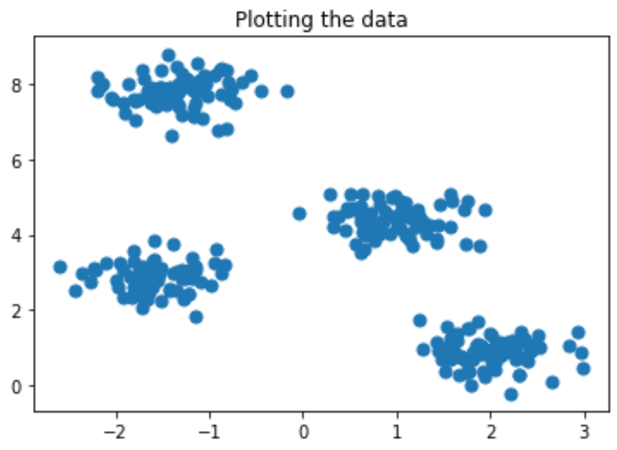
Since we need a distance metric, to measure the distance to the center of the clusters, we can write it down as a helper function pre-hand to aid with calculations, in the implementation below I simply implement the euclidean distance, but many metrics such as the manhattan distance and more distance metrics can be used.

In [2]:
```
def eucl_dist(p1, p2):
    axis_val = 0
    if len(p1.shape) != 1:
        axis_val = 1
    return np.linalg.norm(np.subtract(p1,p2),axis=axis_val)
```

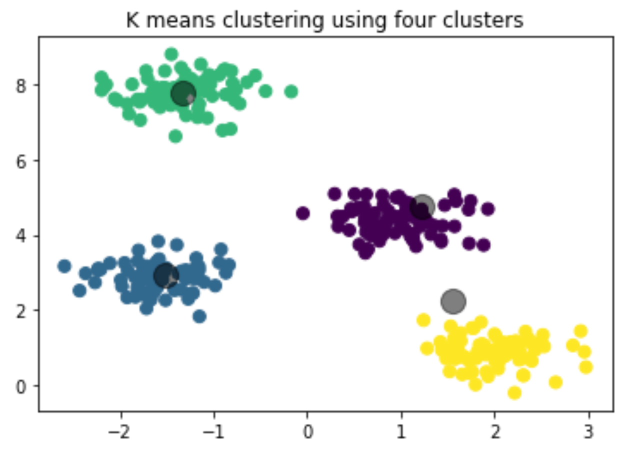
```
In [3]: def k_means(data,k):
            features = data
            dim1 = features.shape[0]
            dim2 = features.shape[1]
            #we will now generate random centers
            mean = np.mean(features, axis = 0)
            std = np.std(features, axis = 0)
            centroids = np.random.randn(k,dim2)*std + mean
            len_centroids= centroids.shape
            centroids_old= np.zeros((len_centroids))
            centroids_new= deepcopy(centroids)
            diff= np.linalg.norm(np.subtract(centroids_new,centroids_old))
            clusters= np.zeros((len_centroids))
            euclidean_distances= np.zeros((dim1,k))

            while diff!=0:
                for num in range(k):
                    euclidean_distances[:,num] = np.linalg.norm(
                        np.subtract(features,centroids[num]),axis=1)
                clusters= np.argmin(euclidean_distances, axis = 1)
                centroids_old= deepcopy(centroids_new)
                for num in range(k):
                    centroids_new[num]= np.mean(features[clusters==num],axis=0)
                diff= np.linalg.norm(np.subtract(centroids_new,centroids_old))
            labels= np.zeros(len(features))
            colors= [0,1,2,3,4,5,6,7,8,9,10,11]
            for i in range(len(features)):
                dist= np.zeros((len(centroids_new)))
                for c in range(len(centroids_new)):
                    dist[c]= np.linalg.norm(np.subtract(features[i],centroids_new[c]),
                                            axis=0)
                index=np.argmin(dist)
                labels[i]= colors[index]
            return labels,centroids_new
```

Let us now load an example dataset

```
In [4]: X, y_true = make_blobs(n_samples=300, centers=4,
                               cluster_std=0.4, random_state=0)
        plt.scatter(X[:, 0], X[:, 1], s=50)
        plt.title("Plotting the data")
        plt.show()
```

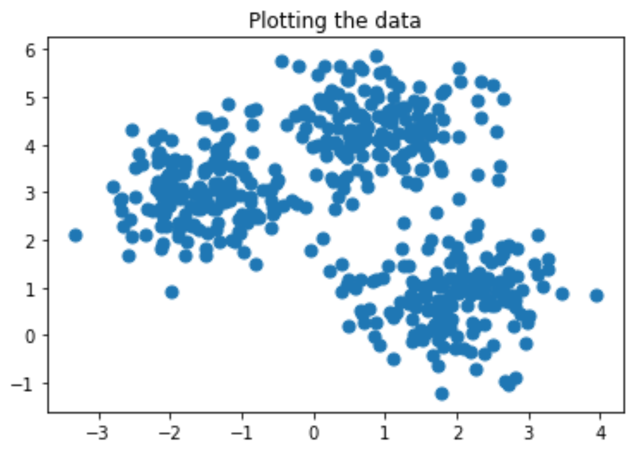


In [6]:
```python
y_predict, centroids =k_means(X,4)
plt.scatter(X[:, 0], X[:, 1], c=y_predict, s=50, cmap='viridis')
plt.scatter(centroids[:, 0], centroids[:, 1], c='black', s=200, alpha=0.5)
plt.title("K means clustering using four clusters")
plt.show()
```

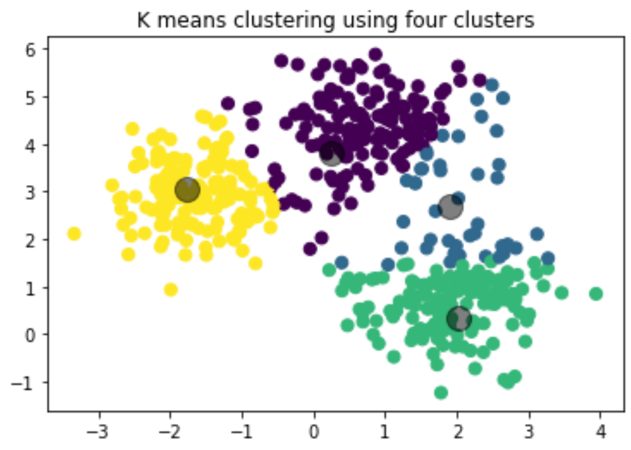


As we can see the algoithm worked! Here since the data was pretty clearly in four clusters, we picked k to be 4, let us now look at a case where the data is more overlapped, and plot a few variations of k, to see which looks best.

In [7]:
```python
X_2, y_true_2 = make_blobs(n_samples=500, centers=3,
                       cluster_std=0.7, random_state=0)
plt.scatter(X_2[:, 0], X_2[:, 1], s=50)
plt.title("Plotting the data")
plt.show()
```

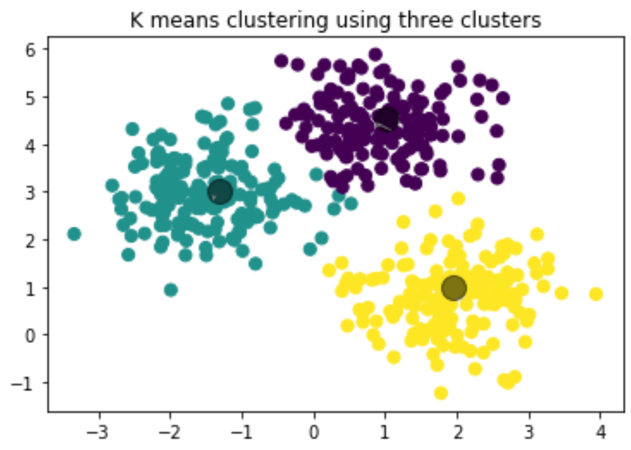


```
In [22]: y_predict_2, centroids_2 =k_means(X_2,4)
         plt.scatter(X_2[:, 0], X_2[:, 1], c=y_predict_2, s=50, cmap='viridis')
         plt.scatter(centroids_2[:, 0], centroids_2[:, 1], c='black', s=200, alpha=0.5)
         plt.title("K means clustering using four clusters")
         plt.show()
```

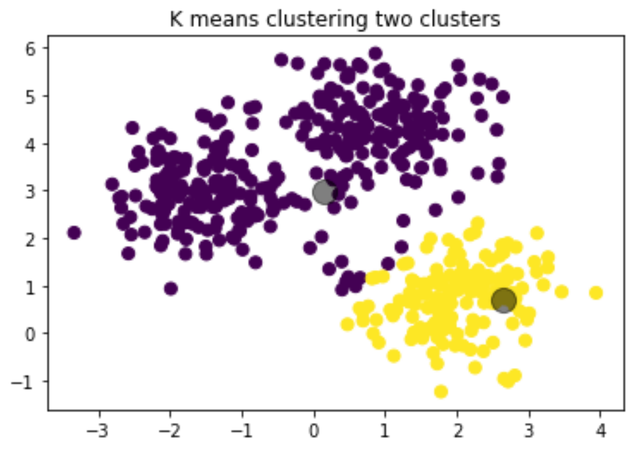


As we can see, using a k value of 4 does not work super well in this case, so let us try another value and see what happens

```
In [16]: y_predict_2, centroids_2 =k_means(X_2,3)
         plt.scatter(X_2[:, 0], X_2[:, 1], c=y_predict_2, s=50, cmap='viridis')
         plt.scatter(centroids_2[:, 0], centroids_2[:, 1], c='black', s=200, alpha=0.5)
         plt.title("K means clustering using three clusters")
         plt.show()
```

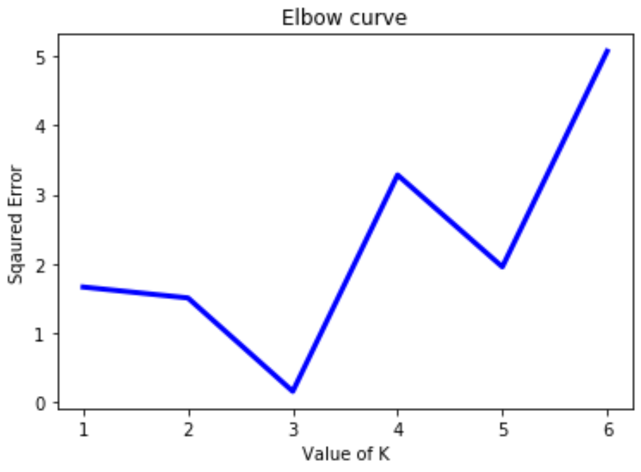


In [17]:
```
y_predict_2, centroids_2 =k_means(X_2,2)
plt.scatter(X_2[:, 0], X_2[:, 1], c=y_predict_2, s=50, cmap='viridis')
plt.scatter(centroids_2[:, 0], centroids_2[:, 1], c='black', s=200, alpha=0.5)
plt.title("K means clustering two clusters")
plt.show()
```



Here we see slightly more distinct clusters shapes in the case of k being equal to 3, but too much overlap in the case of k being equal to 2, is there a way to quantitavely analyze this, without depending on a qualitative analysis?

In [21]:
```
cost =[]
for i in range(1, 7):
    y_predict_2, centroids_2 =k_means(X_2,i)
    cost.append(mean_squared_error(y_true_2, y_predict_2))

plt.plot(range(1, 7), cost, color ='b', linewidth ='3')
plt.xlabel("Value of K")
plt.ylabel("Sqaured Error")
plt.title('Elbow curve')
plt.show()
```



What we see above is an elbow curve, a very common metric to determine the k value in k-means, k nearest neighbours, and many other machine learning methods. It helps quantitatively determine the correct value of k, by picking the value of k that froms the elbow. Even though it may look like a toss up between the values of 3 and 4, it is clear from the elbow curve that 3 is the best value for this particular set.

3.2.1 Examples

K Means is an EM algorithm, which means that the algorithm alternated between two steps, expectation and maximization, till the centroids converge.

K Means is primarily used for unlabeled data, and has many applications such as customer segmentation, as well as insurance fraud.

K Means returns k centroids for the data, and k clusters can be made from the data by assigning each data point to one of the k centroids, whichever is nearest.

Mathematically, K Means alternates between two steps. To initialize, we pick k random points as the k centroids. Then, we determine the cluster partitions - this is done simply by assigning every data point to the nearest centroid.This would be the expectation step. Next, for each of these clusters that are determined, we determine the centroids for these clusters, which is the maximixation step (technically minimizing the sum of distance from the centroid).

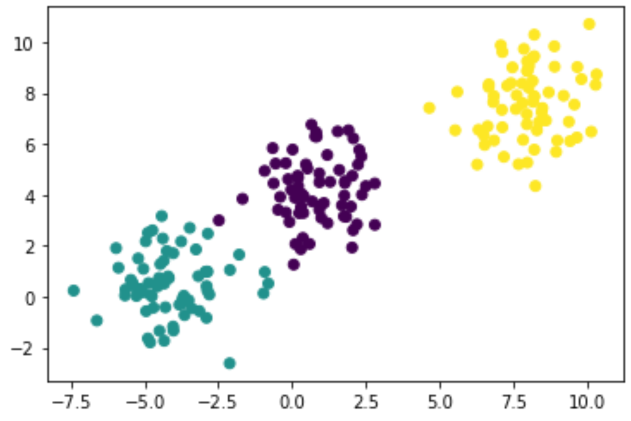
We'll look at an example using scikit-learn's out of the box method for K Means Clustering.

Let's start with importing some necessary modules/packages.

```
In [0]: import numpy as np
        import matplotlib.pyplot as plt
        from sklearn.cluster import KMeans
        %matplotlib inline
```

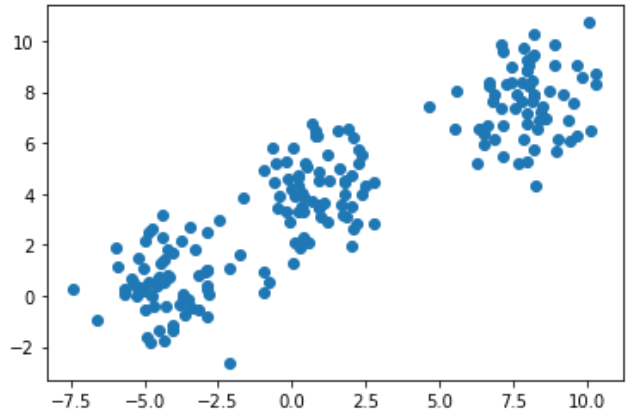
Next let's make some data so we can apply K Means.

In [10]:
```
# Let's make some data
from sklearn.datasets.samples_generator import make_blobs

X, y = make_blobs(n_samples = 200, n_features = 2, centers = 3,
                cluster_std = 1.2, random_state = 3)

plt.scatter(X[:,0], X[:,1], marker = 'o', c = y)
plt.show()
```



This data is colored by blob due to the make_blobs method returning labels for the data. However, we won't use these labels at all for K Means, but can use it at the end to evaluate the algorithm's results.

In [11]:
```
# Here's what we'll be using / how K Means will see it

plt.scatter(X[:,0], X[:,1], marker = 'o')
plt.show()
```



```
In [12]: # The labels to compare to later

         y
```

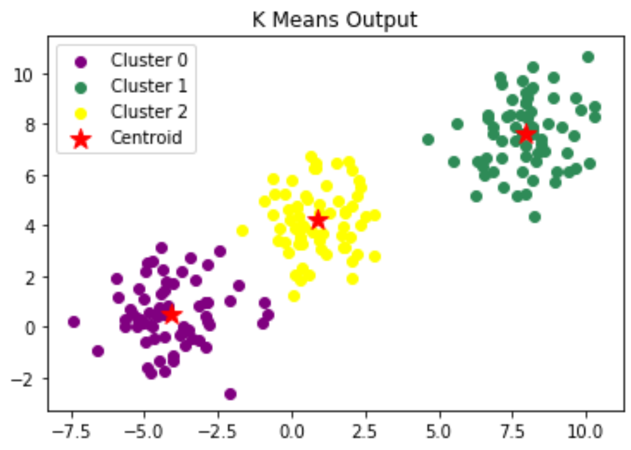
```
Out[12]: array([1, 0, 2, 1, 2, 1, 0, 0, 1, 1, 1, 0, 0, 2, 0, 0, 1, 2, 2, 1, 0, 2,
                2, 2, 0, 1, 1, 0, 0, 0, 0, 0, 2, 0, 1, 0, 1, 0, 1, 2, 1, 1, 1, 0,
                2, 2, 0, 0, 0, 1, 1, 2, 2, 2, 1, 0, 1, 0, 1, 0, 1, 0, 0, 2, 2, 0,
                0, 1, 0, 1, 0, 1, 1, 0, 1, 1, 1, 2, 1, 1, 1, 1, 1, 0, 2, 2, 1, 2,
                1, 2, 2, 2, 2, 2, 2, 0, 2, 1, 0, 0, 0, 2, 1, 2, 1, 1, 0, 0, 2, 2,
                0, 0, 2, 2, 0, 1, 0, 2, 1, 1, 0, 0, 2, 2, 1, 2, 1, 1, 1, 0, 1, 2,
                2, 1, 2, 0, 2, 0, 0, 2, 0, 2, 0, 2, 2, 0, 1, 1, 0, 0, 1, 2, 0, 1,
                0, 1, 2, 2, 1, 1, 2, 2, 2, 0, 2, 1, 0, 0, 2, 1, 1, 0, 2, 1, 2, 0,
                1, 1, 2, 0, 1, 1, 0, 0, 2, 2, 0, 2, 1, 0, 2, 2, 1, 0, 2, 2, 2, 0,
                1, 2])
```

Now let's run K Means

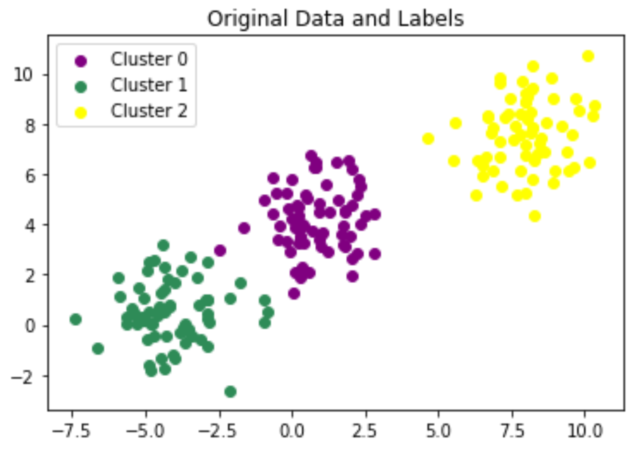
```
In [21]: km = KMeans(n_clusters = 3, max_iter = 100)
         km.fit(X)
```

```
Out[21]: KMeans(algorithm='auto', copy_x=True, init='k-means++', max_iter=100,
                n_clusters=3, n_init=10, n_jobs=None, precompute_distances='auto',
                random_state=None, tol=0.0001, verbose=0)
```

```
In [22]: centroids = km.cluster_centers_
         labels = km.labels_

         plt.scatter(X[labels == 0, 0], X[labels == 0, 1],
                     c='purple', label='Cluster 0')
         plt.scatter(X[labels == 1, 0], X[labels == 1, 1],
                     c='seagreen', label='Cluster 1')
         plt.scatter(X[labels == 2, 0], X[labels == 2, 1],
                     c='yellow', label='Cluster 2')
         plt.scatter(centroids[:, 0], centroids[:, 1], marker='*', c='r', s = 150,
                     label='Centroid')

         plt.legend()
         plt.title("K Means Output")
         plt.show()
```



In [18]:
```python
# Let's look at the original data again and the labels

plt.scatter(X[y == 0, 0], X[y == 0, 1],
            c='purple', label='Cluster 0')
plt.scatter(X[y == 1, 0], X[y == 1, 1],
            c='seagreen', label='Cluster 1')
plt.scatter(X[y == 2, 0], X[y == 2, 1],
            c='yellow', label='Cluster 2')

plt.legend()
plt.title("Original Data and Labels")
plt.show()
```



The K Means result look fairly good, let's evaluate how well they classified the data. As we see above, the labels returned by K Means aren't the same as the original, as the actual label itself is arbitrary. We modify y to more easily compute the accuracy of the K Means output.

In [23]:
```python
# In y, 0 corresponds to 2, 1 to 0, and 2 to 1

y[y==0] = 5
y[y==1] = 0
y[y==2] = 1
y[y==5] = 2

print(y)
```

```
[0 2 1 0 1 0 2 2 0 0 0 2 2 1 2 2 0 1 1 0 2 1 1 1 2 0 0 2 2 2 2 2 1 2 0 2 0
 2 0 1 0 0 0 2 1 1 2 2 2 0 0 1 1 1 0 2 0 2 0 2 0 2 2 1 1 2 2 0 2 0 2 0 0 2
 0 0 0 1 0 0 0 0 0 2 1 1 0 1 0 1 1 1 1 1 1 2 1 0 2 2 2 1 0 1 0 0 2 2 1 1 2
 2 1 1 2 0 2 1 0 0 2 2 1 1 0 1 0 0 0 2 0 1 1 0 1 2 1 2 2 1 2 1 2 1 1 2 0 0
 2 2 0 1 2 0 2 0 1 1 0 0 1 1 1 2 1 0 2 2 1 0 0 2 1 0 1 2 0 0 1 2 0 0 2 2 1
 1 2 1 0 2 1 1 0 2 1 1 1 2 0 1]
```

In [24]:
```python
# Find accuracy

accuracy = (np.sum(labels == y) / len(y)) * 100
print("Accuracy of K Means for this data: " + str(accuracy) + "%")
```

```
Accuracy of K Means for this data: 99.5%
```

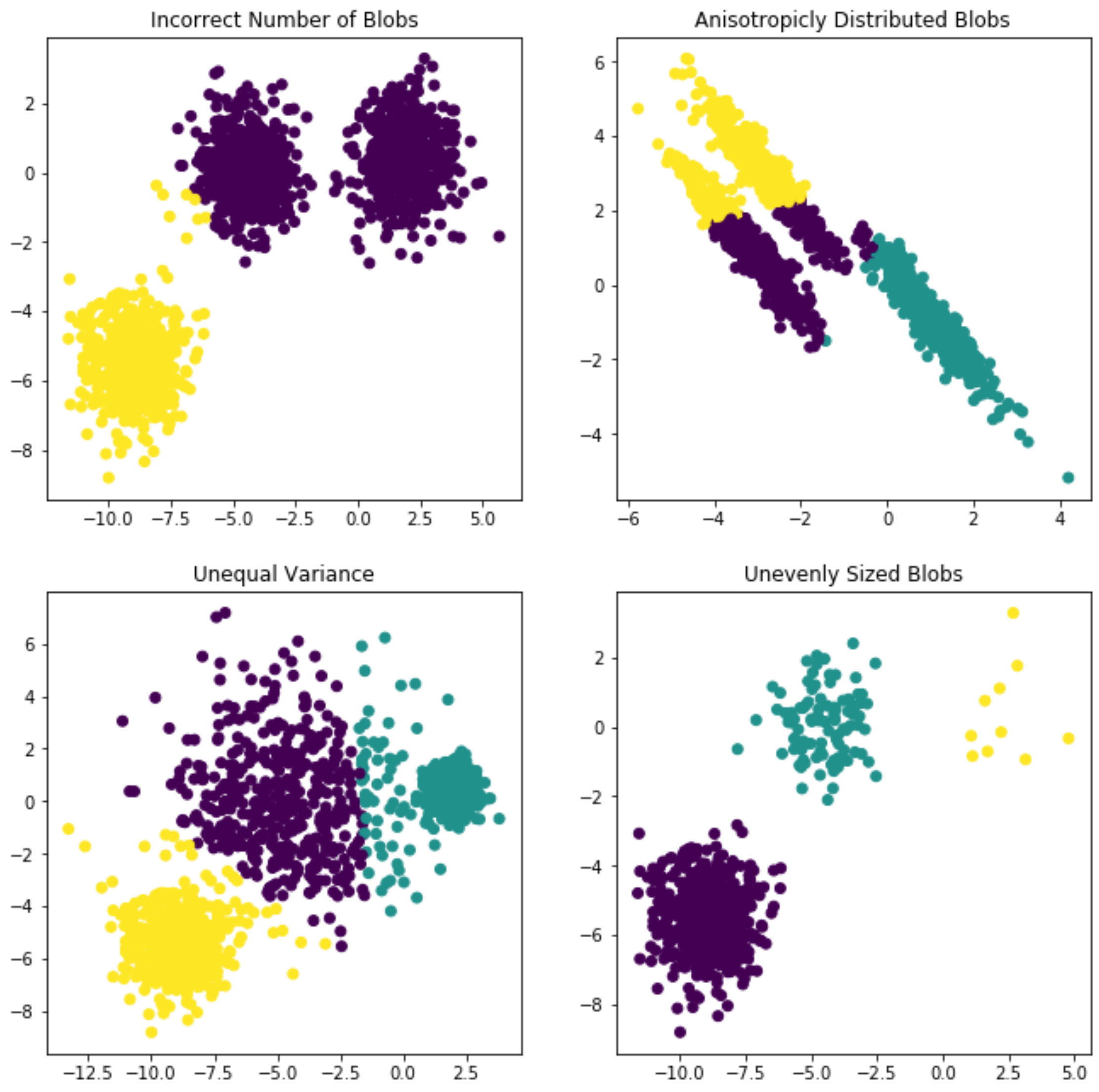
So overall, it did really well! Let's also consider some common pitfalls of K Means Clustering via some examples.

Common Pitfalls of K Means Clustering

The following examples show how issues such as incorrect value for k (as it is a hyperparameter the algorithm doesn't set the correct k), anisotropicly distributed clusters, and unequal variance amongst clusters. Another scenario, different sizes for the clusters, doesn't break K Means and the algorithm returns good clusters for the data.

All of these can be seen below:

In [25]:
```
# Run K Means on some data that doesn't follow some assumptions
# Then plot it all

plt.figure(figsize=(11, 11))

n_samples = 1500
random_state = 170
X, y = make_blobs(n_samples=n_samples, random_state=random_state)

# Incorrect number of clusters
y_pred = KMeans(n_clusters=2, random_state=random_state).fit_predict(X)

plt.subplot(221)
plt.scatter(X[:, 0], X[:, 1], c=y_pred)
plt.title("Incorrect Number of Blobs")

# Anisotropicly distributed data
transformation = [[0.60834549, -0.63667341], [-0.40887718, 0.85253229]]
X_aniso = np.dot(X, transformation)
y_pred = KMeans(n_clusters=3, random_state=random_state).fit_predict(X_aniso)

plt.subplot(222)
plt.scatter(X_aniso[:, 0], X_aniso[:, 1], c=y_pred)
plt.title("Anisotropicly Distributed Blobs")

# Different variance
X_varied, y_varied = make_blobs(n_samples=n_samples,
                                cluster_std=[1.0, 2.5, 0.5],
                                random_state=random_state)
y_pred = KMeans(n_clusters=3, random_state=random_state).fit_predict(X_varied)

plt.subplot(223)
plt.scatter(X_varied[:, 0], X_varied[:, 1], c=y_pred)
plt.title("Unequal Variance")

# Unevenly sized blobs
X_filtered = np.vstack((X[y == 0][:500], X[y == 1][:100], X[y == 2][:10]))
y_pred = KMeans(n_clusters=3,
                random_state=random_state).fit_predict(X_filtered)

plt.subplot(224)
plt.scatter(X_filtered[:, 0], X_filtered[:, 1], c=y_pred)
plt.title("Unevenly Sized Blobs")

plt.show()
```



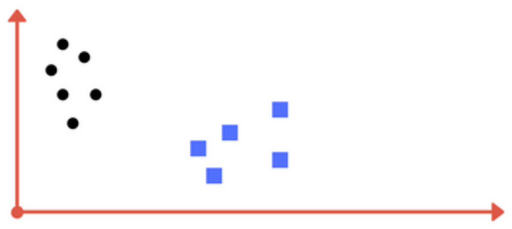
Credit:

K Means Assumptions Visual - Scikit-Learn https://scikit-learn.org/stable/auto_examples/cluster/plot_kmeans_assumptions.html#sphx-glr-auto-examples-cluster-plot-kmeans-assumptions-py (https://scikit-learn.org/stable/auto_examples/cluster/plot_kmeans_assumptions.html#sphx-glr-auto-examples-cluster-plot-kmeans-assumptions-py)

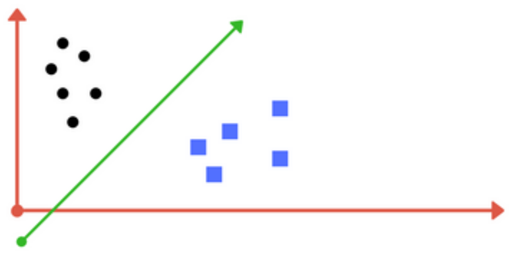
3.3 Support Vector Machines

SVMs are a tool used for classifying data. The goal of an SVM is to come up with decision boundary to classify the training data based on the labels. Geometrically, we consider the dimensions to refer to the features of the data, and SVM determines a hyperplane to divide the training data points such that the two sides correspond to the different labels.

Starting Point for Training Data



Hyperplane Decision Boundary



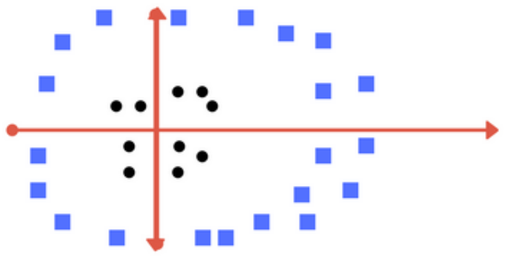
The decision boundary found by an SVM is the hyperplane that yields the max margin between the 2 sets of training data points. There are also a few other cases applications to consider for SVMs.

Kernels

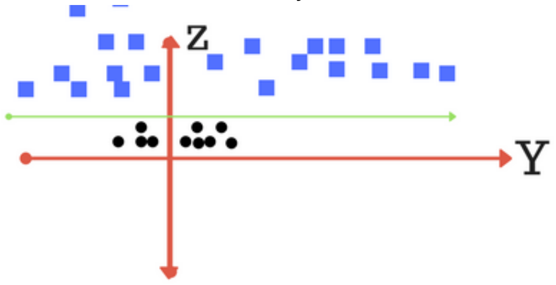
To start with, using a kernel on our features is a common trick that helps extend simple tools to a wide variety of models. 2 common kernels are gaussian (RDF) and polynomial.

To look at polynomial as an example, we can see an efficient transformation on the following data that allows us to use SVMs, which create a hyperplane (linear decision boundary).

Dataset with no linear decision boundary

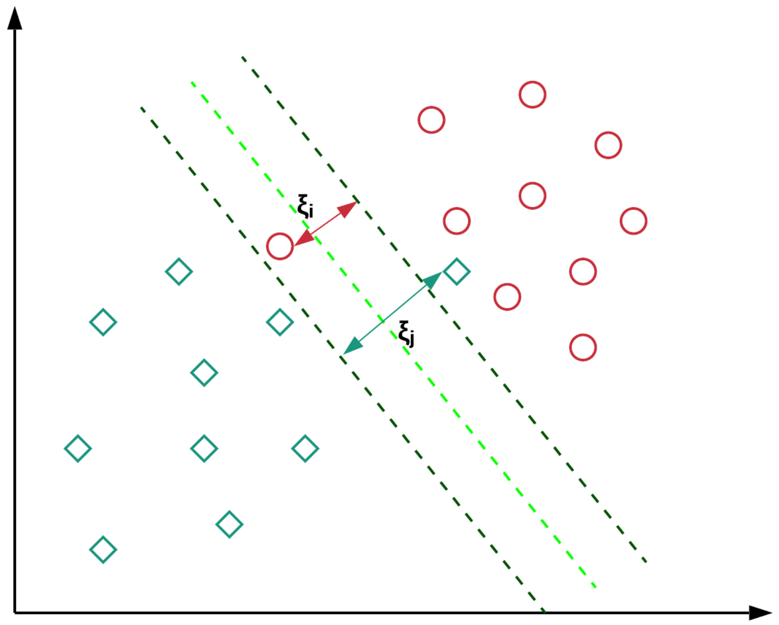


Kernelizing the data yields a linear decision boundary.



Hard vs Soft SVM

In the visuals above, all of the data was linearly separable - there was a line neatly dividing the 2 sets. In the case that the data isn't linearly separable, we essentially add a slack variable to our constraints to allow for some (few) points to fall on the wrong side of the decision boundary.



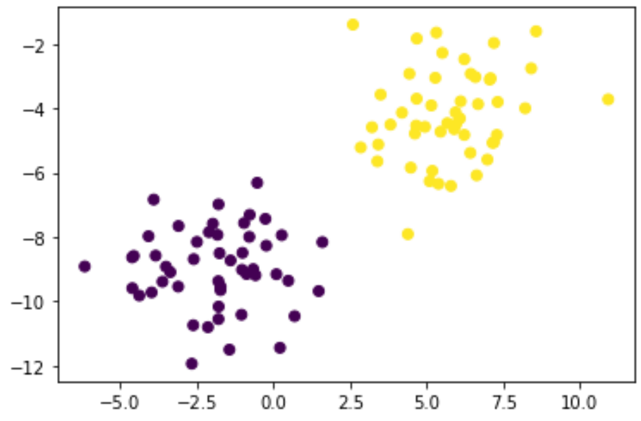
In order to make sure that this slack is also minimized, we also add it to the cost we are trying to minimize along with C, a hyperparameter multiplied to the slack.

	small C	large C
Desire	maximize margin	keep slack small or zero
Danger	underfitting	overfitting
Outliers	less sensitive	more sensitive

SVM Example

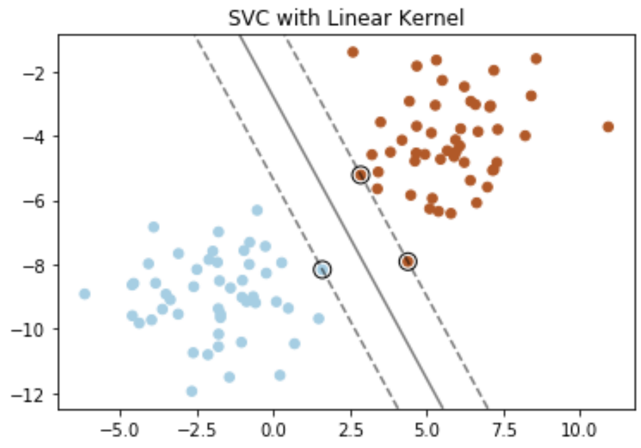
```
In [0]: import numpy as np
        import matplotlib.pyplot as plt
        from sklearn import svm
```

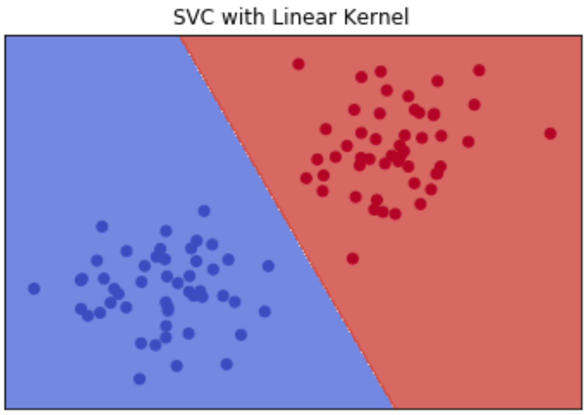
In [0]:
```
# Let's make some data
from sklearn.datasets.samples_generator import make_blobs
X, y = make_blobs(n_samples = 100, n_features = 2, centers = 2,
                  cluster_std = 1.5, random_state = 40)

def plot_raw_data(X,y):
    plt.scatter(X[:,0], X[:,1], marker = 'o', c = y)
    plt.show()

plot_raw_data(X,y)
```



In [0]:
```
# For SVM we treat labels as 1 or -1, so we need to fix our labels
# Using the sklearn svm this isn't necessary however

svc = svm.SVC(kernel = "linear").fit(X, y)
```

In [0]:
```python
# Plot the decision boundary of the SVM

def plot_boundary(clf, X, y, clf_name):

    plt.scatter(X[:, 0], X[:, 1], c=y, s=30, cmap=plt.cm.Paired)

    # plot the decision function
    ax = plt.gca()
    xlim = ax.get_xlim()
    ylim = ax.get_ylim()

    # create grid to evaluate model
    xx = np.linspace(xlim[0], xlim[1], 30)
    yy = np.linspace(ylim[0], ylim[1], 30)
    YY, XX = np.meshgrid(yy, xx)
    xy = np.vstack([XX.ravel(), YY.ravel()]).T
    Z = clf.decision_function(xy).reshape(XX.shape)

    # plot decision boundary and margins
    ax.contour(XX, YY, Z, colors='k', levels=[-1, 0, 1], alpha=0.5,
            linestyles=['--', '-', '--'])
    # plot support vectors
    ax.scatter(clf.support_vectors_[:, 0], clf.support_vectors_[:, 1], s=100,
            linewidth=1, facecolors='none', edgecolors='k')
    plt.title(clf_name)
    plt.show()


def plot_regions(clf, X, y, clf_name):
    # step size in mesh
    h = 0.02

    # create a mesh to plot in
    x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
                         np.arange(y_min, y_max, h))

    Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])

    # Put the result into a color plot
    Z = Z.reshape(xx.shape)
    plt.contourf(xx, yy, Z, cmap=plt.cm.coolwarm, alpha=0.8)

    # Plot also the training points
    plt.scatter(X[:, 0], X[:, 1], c=y, cmap=plt.cm.coolwarm)
    plt.xlim(xx.min(), xx.max())
    plt.ylim(yy.min(), yy.max())
    plt.xticks(())
    plt.yticks(())
    plt.title(clf_name)
    plt.show()

plot_boundary(svc, X, y, "SVC with Linear Kernel")
plot_regions(svc, X, y, "SVC with Linear Kernel")
```

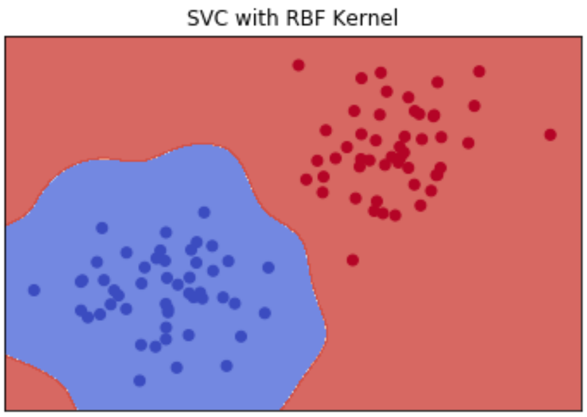




We can also visualize what a gaussian kernel would make the decision boundary look like.

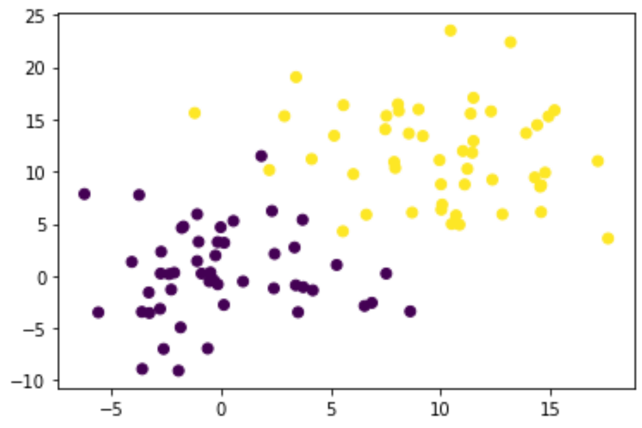
```
In [0]: svc_RBF = svm.SVC(kernel = "rbf", gamma=0.7).fit(X, y)
```

```
In [0]: plot_regions(svc_RBF, X, y, "SVC with RBF Kernel")
```



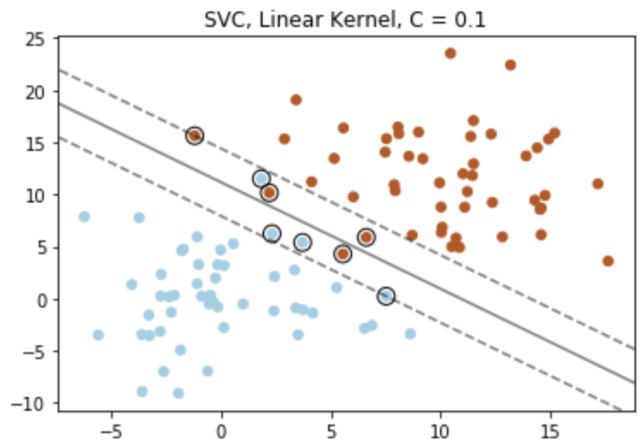
Another aspect we can explore is changing C, a hyperparameter, when we run SVM.

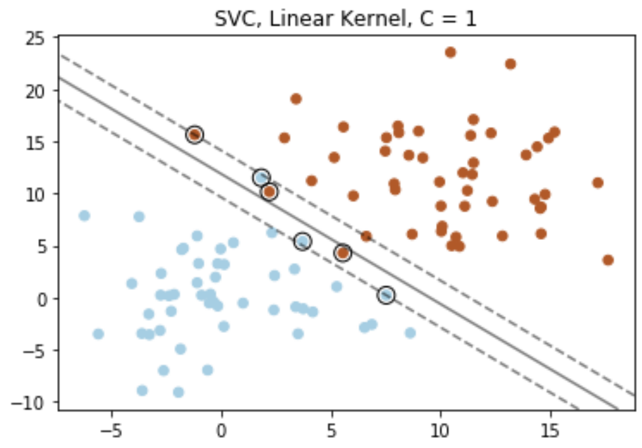
```
In [0]: # Generate new data that is more noisy

        X2, y2 = make_blobs(n_samples = 100, n_features = 2, centers = [[0,0],[12,12]],
                        cluster_std = 4)
        plot_raw_data(X2,y2)
```

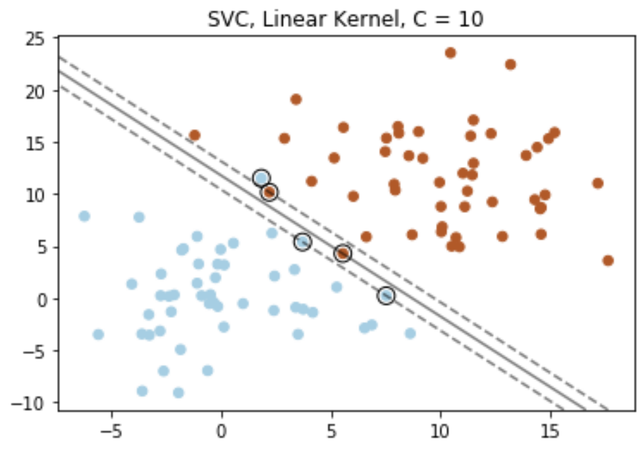


```
In [0]: # Let's run SVM with varying C values
        c_vals = [0.1, 1, 10]
        plot_titles = ["SVC, Linear Kernel, C = " + str(i) for i in c_vals]
```

```
In [0]: for i in range(len(c_vals)):
            plot_boundary(svm.SVC(kernel = "linear", C = c_vals[i]).fit(X2, y2), X2,
                          y2, plot_titles[i])
```







We see in the original data that there is a purple dot that we can consider an outlier. For low C, the outlier doesn't skew the decision boundary, while in the higher C value, the SVM is more sensitive to the outlier.

We also see that for smaller C, the margin is larger while for the larger C the margin is much smaller.

Credit for visualization (sklearn documentation):

- https://scikit-learn.org/0.18/auto_examples/svm/plot_iris.html (https://scikit-learn.org/0.18/auto_examples/svm/plot_iris.html)
- https://scikit-learn.org/stable/auto_examples/svm/plot_separating_hyperplane.html (https://scikit-learn.org/stable/auto_examples/svm/plot_separating_hyperplane.html)

Images:

- https://medium.com/machine-learning-101/chapter-2-svm-support-vector-machine-theory-f0812effc72 (https://medium.com/machine-learning-101/chapter-2-svm-support-vector-machine-theory-f0812effc72)

3.3.1 Implementation

SVMs, as covered in the intro notebook, find a hyperplane that maximizes the margin between 2 sets.

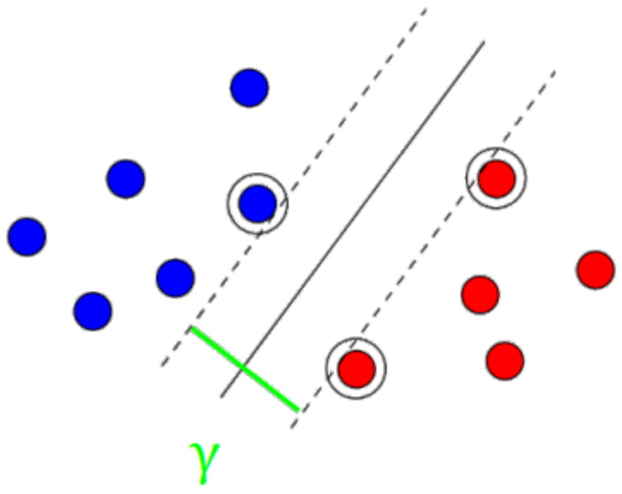
Even with more than 2 labels/groups, the most common SVM processes compare 2 sets at a time to determine a hyperplane, with either taking 2 at a time or a one vs the rest approach.

The Problem

Mathematically, we can come up with an objective that we are trying to optimize. In this case, we have a margin *m* to maximize.

```
In [0]: from IPython.display import Image
        Image(filename="/content/data/margin.PNG")
```

```
Out[0]:
```



Here we'll go through code for the hard margin formulation, which doesn't allow for points to fall within the margin of the hyperplane.

Based on this definition, while we're maximizing m, our constraint is that the points must fall outside of the hyperplane and margin. Say *w* is vector normal to the hyperplane, then we can define the distance as follows:

```
In [0]: Image(filename="/content/data/distance.PNG")
```

```
Out[0]:
```

$$\frac{|\mathbf{w}^\top \mathbf{z} - b|}{\|\mathbf{w}\|_2}$$

Here, the plane H is defined as all x such that w^T * x - b = 0. The distance uses z as a point not in H, the hyperplane.

The constraint is that for all point in the given data sets, distance from the hyperplane must be greater than m.

```
In [0]: Image(filename="/content/data/constraint.PNG")
```

Out[0]:

$$y_i \frac{(\mathbf{w}^\top \mathbf{x}_i - b)}{\|\mathbf{w}\|_2} \geq m$$

The y_i is multiplied to make the numerator positive. This is also the reason our labels for this problem should be +1 or -1 for the 2 different groups.

Finally with some mathematical manipulation, we can take norm(w) = 1/m, and come up with the following final formulation for this problem.

```
In [0]: Image(filename="/content/data/problem.PNG")
```

Out[0]:

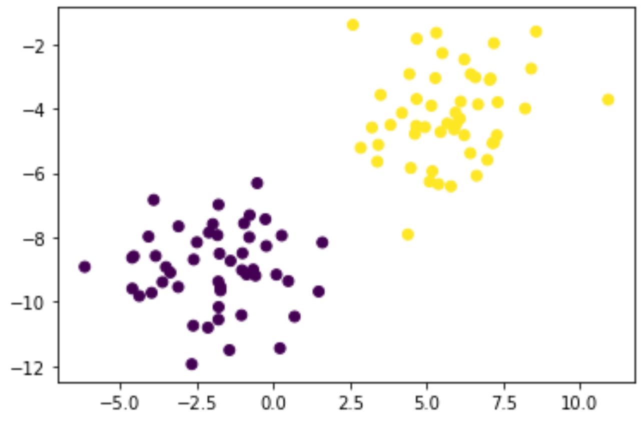
$$\boxed{\begin{array}{ll} \min\limits_{\mathbf{w},b} & \frac{1}{2}\|\mathbf{w}\|_2^2 \\ \text{s.t.} & y_i(\mathbf{w}^\top \mathbf{x}_i - b) \geq 1 \quad \forall i \end{array}}$$

Solving Optimization Problems

The problem above is a quadratic problem, and this problem is convex. What this means is that we're able to solve the problem with various techniques. We can also use cvxpy, a package specifically for solving convex optimization problems.

First, let's generate some data and visualize it:

In [0]:
```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets.samples_generator import make_blobs
X, y = make_blobs(n_samples = 100, n_features = 2, centers = 2,
                cluster_std = 1.5, random_state = 40)

def plot_raw_data(X,y):
    plt.scatter(X[:,0], X[:,1], marker = 'o', c = y)
    plt.show()

plot_raw_data(X,y)
```



In [0]:
```python
# Need to turn y to +1 and -1, currently +1 and 0
y[y == 0] = -1
print(y)
```

```
[ 1 -1  1  1 -1 -1 -1  1  1 -1 -1  1  1  1  1  1  1  1 -1 -1  1 -1  1  1
 -1  1 -1  1  1 -1 -1 -1  1  1  1 -1 -1 -1 -1  1 -1 -1  1 -1  1 -1  1 -1
 -1  1  1  1 -1  1  1  1 -1 -1  1 -1  1  1  1 -1 -1 -1  1  1 -1  1 -1 -1
 -1  1  1 -1  1 -1  1 -1  1  1  1 -1 -1 -1 -1 -1  1  1 -1 -1  1  1 -1  1
 -1 -1 -1 -1]
```

In [0]:
```python
import cvxpy as cp

w = cp.Variable(2)
b = cp.Variable(1)
```

Next we create our objective and the constraints with the 'variables' above.

Note: The cp.Variables refer to the actual variables we're optimizing in the problem.

In [0]:
```python
ob = cp.Minimize(1/2*(w[0]**2+w[1]**2))
co = []
for i in range(len(X)):
    co.append(y[i]*(w[0]*X[i][0] + w[1]*X[i][1] - b) >= 1)
```

In [0]:
```python
prob = cp.Problem(ob, co)
prob.solve()
# Returns the value of the objective after solving for the minimum
```

Out[0]:
```
0.3078836992495939
```

```
In [0]: print("Optimal w: " + str(w.value))
        print("Optimal b: " + str(b.value))
```

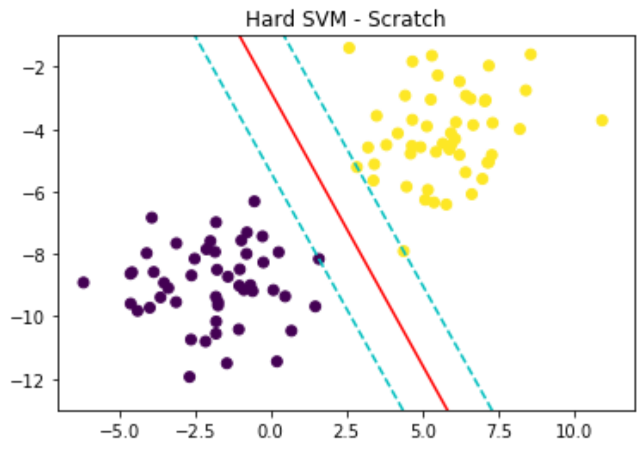
```
Optimal w: [0.68148252 0.38903596]
Optimal b: [-1.08908035]
```

We can plot our solution and the margin using the original formula:

w^T * x - b = 0

And we know the margins are where the above term is equal to +1 and -1.

```
In [0]: # Plot the solution with the original data
        x = np.linspace(-7.5, 12, 1000)
        plt.plot(x, (b.value[0] - w.value[0]*x)/w.value[1], "-r")

        # Dashed refers to the decision boundary + margin
        plt.plot(x, (1 + b.value[0] - w.value[0]*x)/w.value[1], "--c")
        plt.plot(x, (-1 + b.value[0] - w.value[0]*x)/w.value[1], "--c")

        plt.ylim(-13,-1)
        plt.xlim(-7,12)
        plt.title("Hard SVM - Scratch")
        plot_raw_data(X,y)
```



Short Note on Soft SVM

For Soft SVM, the issue lies in data that isn't neatly linearly separable.

For this, we must relax the constraint that every data point must be outside of the the decision boundary +- margin. Instead we allow some points to fall within/across this by introducing an error as a slack variable.

We also want to minimize this error to have few points fall across the decision boundary, and so we incorporate it into the decision boundary.

```
In [0]: Image(filename="/content/data/soft.PNG")
```

Out[0]:

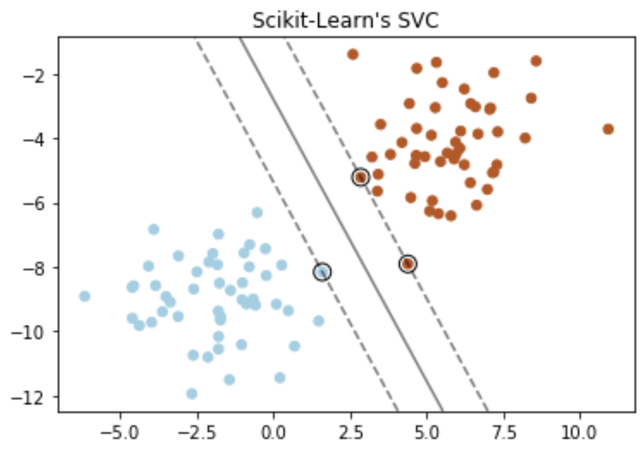
$$
\begin{aligned}
\min_{\mathbf{w},b,\xi_i} \quad & \frac{1}{2}\|\mathbf{w}\|^2 + C\sum_{i=1}^{n}\xi_i \\
\text{s.t.} \quad & y_i(\mathbf{w}^\top \mathbf{x}_i - b) \geq 1 - \xi_i \quad \forall i \\
& \xi_i \geq 0 \quad \forall i
\end{aligned}
$$

Comparison to Scikit-Learn Implementation

The scikit-learn SVM implementation uses the above soft SVM formulation. As we see above when the data is linearly separable, we can easily set our error to be 0, simplifying to the hard SVM problem.

Let's compare our solution to scikit's out-of-the-box SVM solver:

```
In [0]: from sklearn import svm

        svc = svm.SVC(kernel = "linear").fit(X, y)

        def plot_boundary(clf, X, y, clf_name):

            plt.scatter(X[:, 0], X[:, 1], c=y, s=30, cmap=plt.cm.Paired)

            # plot the decision function
            ax = plt.gca()
            xlim = ax.get_xlim()
            ylim = ax.get_ylim()

            # create grid to evaluate model
            xx = np.linspace(xlim[0], xlim[1], 30)
            yy = np.linspace(ylim[0], ylim[1], 30)
            YY, XX = np.meshgrid(yy, xx)
            xy = np.vstack([XX.ravel(), YY.ravel()]).T
            Z = clf.decision_function(xy).reshape(XX.shape)

            # plot decision boundary and margins
            ax.contour(XX, YY, Z, colors='k', levels=[-1, 0, 1], alpha=0.5,
                    linestyles=['--', '-', '--'])
            # plot support vectors
            ax.scatter(clf.support_vectors_[:, 0], clf.support_vectors_[:, 1], s=100,
                    linewidth=1, facecolors='none', edgecolors='k')
            plt.title(clf_name)
            plt.show()

        plot_boundary(svc, X, y, "Scikit-Learn's SVC")
```



Credit for proofs: EECS 189 Course Notes