5. Supervised Learning

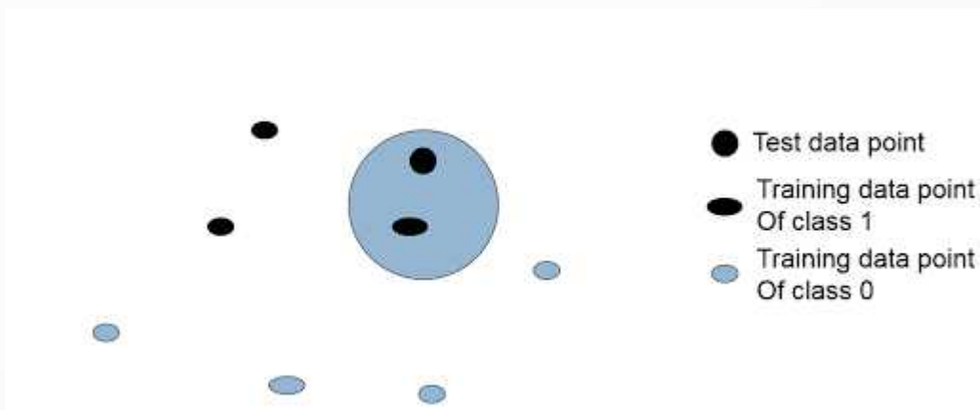
KNN
SVM
Linear SVM
Non-linear SVM

KNN

Algorithm & Variants
Best K

KNN
Algorithm & Variants

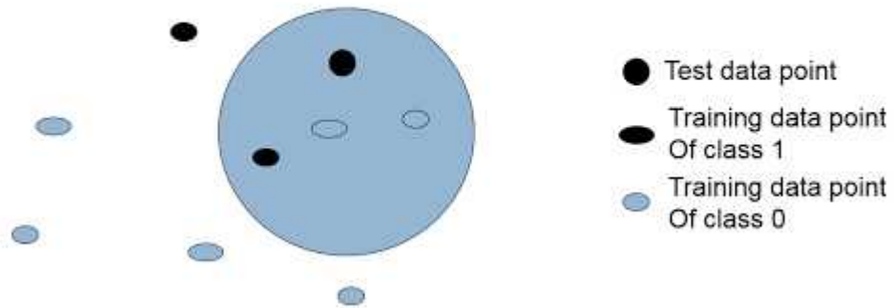
- For both classification and regression
  - A useful technique is to assign weights based on the neighbours.
  - The nearer neighbors contribute more to the average than the more distant ones.
- The basic idea is to label the test data point is the same as the nearest neighbor.



- If a black circle as a test point falls into a region in which the closest point is a black ellipse with class label of 1, based on the nearest neighbor, we are going to label this new sample as class 1.

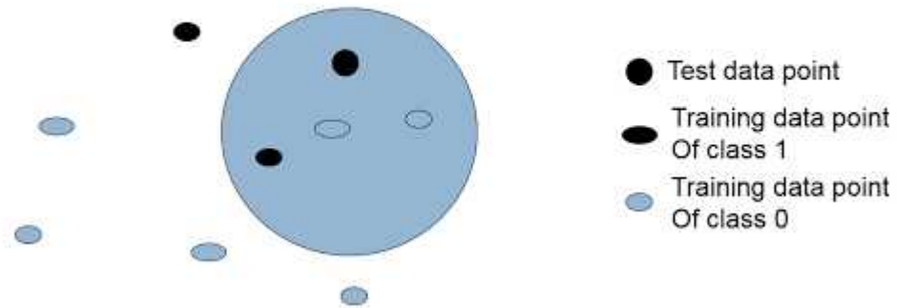
KNN
Algorithm & Variants...

- But also K in KNN can vary.



- Lets say someone would like to check K nearest neighbours of the test point in order to make the decision. So you can label a test instance same as the majority label of the K-nearest neighbours.
- The figure is an example a of 3−NN classification.

KNN
Algorithm & Variants...



- As you saw in the figure, a new test point falls into the scope of two training points from class 0 and one from class 1.
- Obviously you trust the 0 class due to majority.

KNN
Best number of neighbors (K)

- How do you pick the variable K which holds the number of neighbours?
- How important is selecting the right K?
- You can think of K as controlling the shape of the decision boundary we talked about earlier.
- For small values of K, we are restraining the region of a given prediction and forcing our classifier to be more focused on the close regions and neighbours.
- We are asking the classifier not to care about fairly distant points.
- This will result in a low bias and high variance (why?)

KNN
Best number of neighbors (K)...

- Higher values of K will have smoother decision boundaries which means lower variance but increased bias.
- So basically, higher values of K means asking for more and more information even from distant training points.
- Like most of machine learning problems, finding hyperparameters such as K is not really straightforward.

KNN
Best number of neighbors (K)...

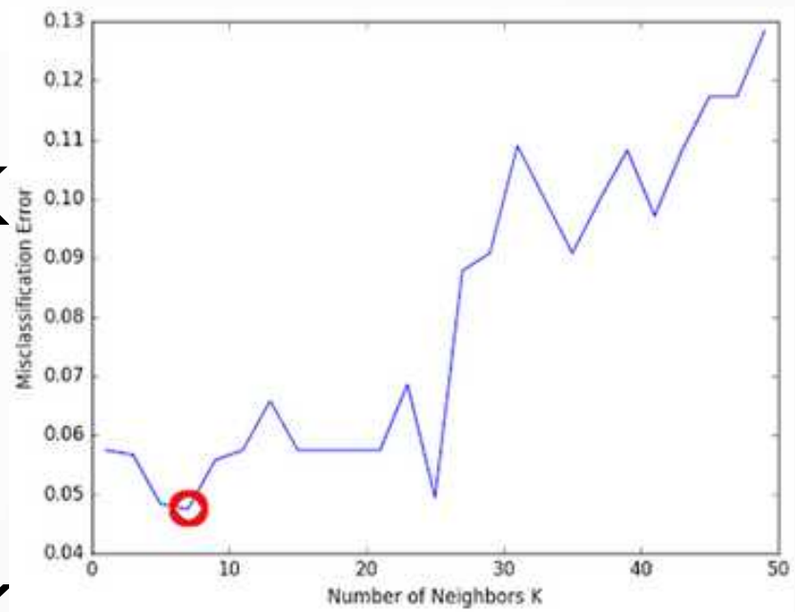
- Finding the best answer is not always possible.
- But as a simple and handy method, you can use Cross-validation (see week 2 of course 3, Model Selection) to partition your data into test and training samples and evaluate your model with different ranges of K values.
- For example you can consider the number of neighbours to be $K = 1, .., K_{max}$
- Now perform the Cross-validation for every possible number of K and evaluate the model based on the training and test data you have partitioned.

KNN
Best number of neighbors (K)...

- You can use the misclassification error as a measurement of performance in your models.
- Finally, by exploring different values of K and their corresponding misclassification error, we can decide which K has the best performance based on our partitioned data.
- You can define K_{max} based on your training data points.
- There is no rule of thumb in selecting K_{max} since it depends on your desired rate of exploration for K

KNN
Best number of neighbors (K)...

- For example when you are dealing with 1000 training data points, you may want to define K=1,...,50
- This figure illustrates a sample plot of Number of Neighbours (K) and misclassification error.



- As you can see the minimum value of the error occurs when K=7.

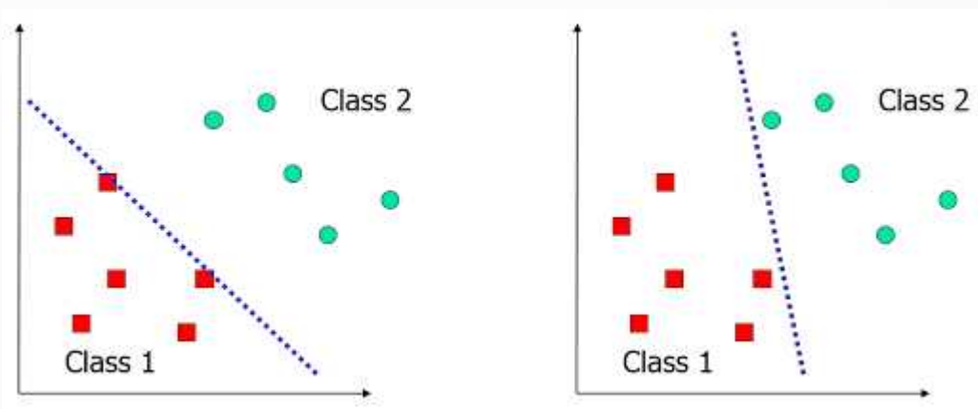
Support Vector Machine (SVM)

Support Vector Machine (SVM)

- SVMs can represent non-linear functions and they have an efficient training algorithm.
- With sufficiently large training data, SVMs can achieve high classification accuracy.
- For example, ≈99% accuracy for handwritten digits recognition.

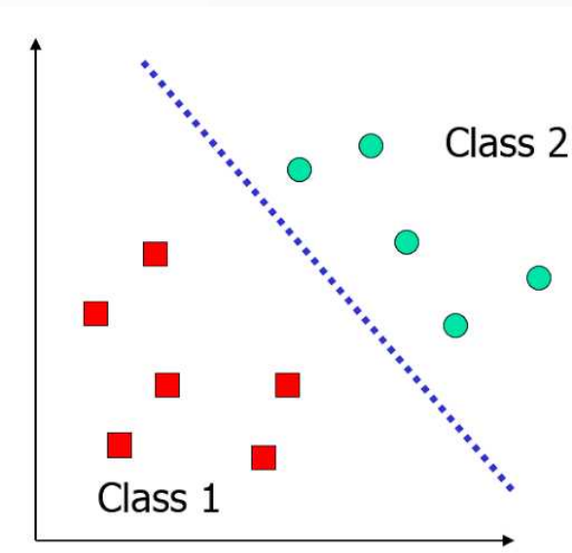
Support Vector Machine (SVM)
Revisiting linear binary classification...

- Consider the following figure as an illustration of boundary in a two class binary classification problem.
- Many decision boundaries can separate these two classes as you can see in the figure.
- Which one should we choose?



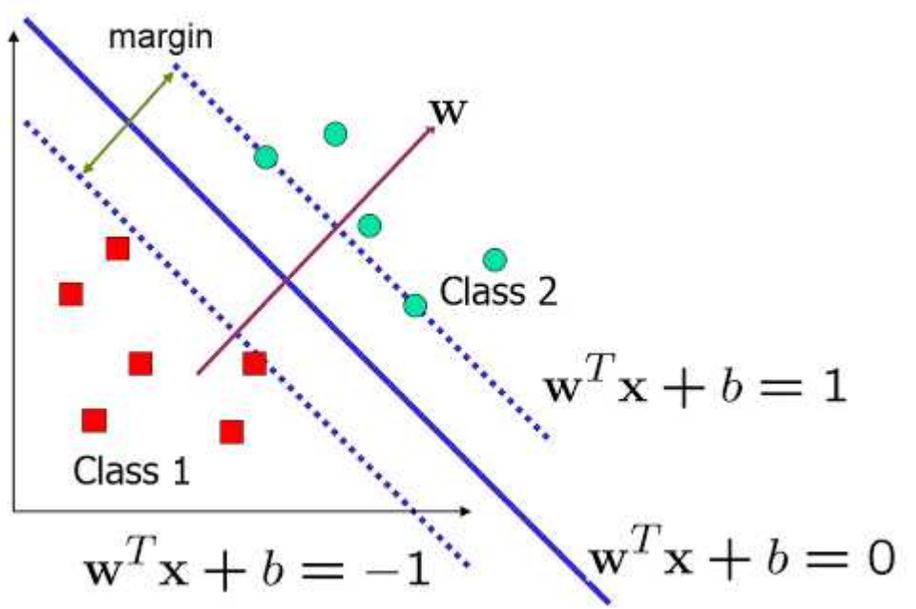
Support Vector Machine (SVM)
Revisiting linear binary classification...

- In SVM, given the labelled data, we do not choose this line randomly.
- We aim for maximizing the margin for this boundary line.
- Consider the figure as another example.
- Can you see the difference now?
- The proposed line looks to be in the middle of the data points with different classes.
- That will lead us to the next topic about margins in SVM.



Support Vector Machine (SVM)
Concept of margin

- The basic idea is that the decision boundary should be as far away from the data of both classes as possible.
- The aim is to minimise possible conflicts which may happen during classification.
- So we should maximise the margin (see figure).

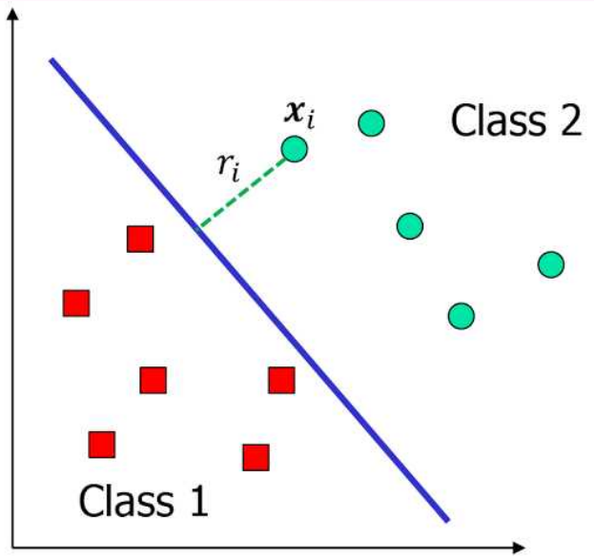


Support Vector Machine (SVM)
Concept of margin...

- As you can see in the figure, we have three lines:
  - $\mathbf{w}^T\mathbf{x} + b = -1$ as the border line of class 1.
  - $\mathbf{w}^T\mathbf{x} + b = 1$ as the border lines of class 2
  - $\mathbf{w}^T\mathbf{x} + b = 0$ as the boundary with a specific margin

- Now let us denote our training instances as $\{\mathbf{x}_i, y_i\}, \quad i = 1, ..., n$ where $y_i \in \{-1,1\}$.
- What is the Euclidean distance from a point x to the decision boundary?

Support Vector Machine (SVM)
Concept of margin...



- In the above figure, we have denoted this distance with .
- We know that the shortest distance between a point and a hyperplane is perpendicular to the plane, and hence, parallel to
- So the distance of to the separating hyperplane is :

$$r_i = y_i \frac{\mathbf{w}^T \mathbf{x}_i + b}{\|\mathbf{w}\|}$$

Support Vector Machine (SVM)
Concept of margin...

- Let us assume that the minimum distance of the hyperplane from any instance is .
- Thus for each instance we have: $r_i = y_i \frac{\mathbf{w}^T \mathbf{x}_i + b}{\|\mathbf{w}\|} \geq r$
- Instances that are closest to the hyperplane are called support vectors and are at distance from the hyperplane.
- So the margin is defined as the distance between the support vectors and is given as $m = 2r = \frac{2}{\|\mathbf{w}\|}$ (why?)

Support Vector Machine (SVM)
Concept of margin...

- Because we would like to have equal distances for both sides of data points ().
- Thus we have $y_i(\mathbf{w}^T\mathbf{x}_i + b) \geq 1$
- we know: $r_i = y_i \frac{\mathbf{w}^T\mathbf{x}_i + b}{\|\mathbf{w}\|} \geq r$
- So we know that $2r = \frac{2}{\|\mathbf{w}\|}$ then $r = \frac{1}{\mathbf{w}}$
- Then, $y_i \frac{\mathbf{w}^T\mathbf{x}_i + b}{\|\mathbf{w}\|} \geq \frac{1}{\|\mathbf{w}\|} = y_i(\mathbf{w}^T\mathbf{x}_i + b) \geq 1$

Linear SVM
SVM formulation (linearly separable data)

- SVM aims to find a hyperplane (w,b) so that the margin $\frac{2}{\|\mathbf{w}\|}$ is maximised while satisfying the constraint $y_i(\mathbf{w}^T\mathbf{x}_i + b) \geq 1$
- SVM formulation therefore solves the following optimisation problem:

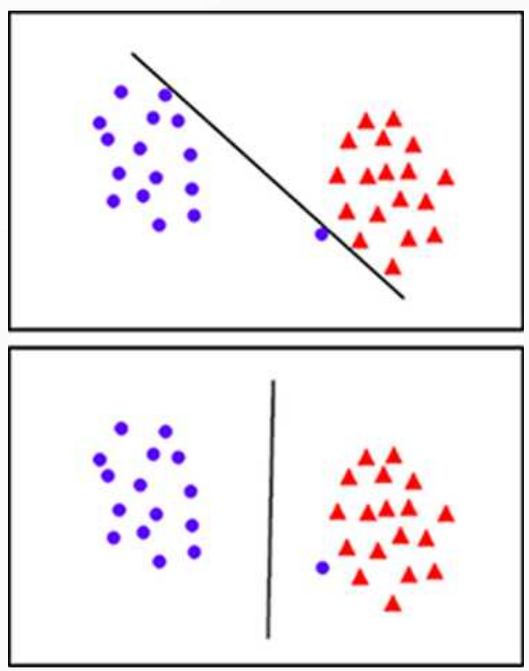
$$Minimize\ \frac{1}{2}\|\mathbf{w}\|^2$$

$$\text{subject to}\ \ \mathrm{y}_i(\mathbf{w}^T\mathbf{x}_i + b) \geq 1 \quad \forall i$$

- Remember the problem of maximising $\frac{2}{\|\mathbf{w}\|}$ is the same as minimising $\frac{1}{2}\|\mathbf{w}\|^2$
- We need to optimise a quadratic function in subject to linear constraints.

Linear SVM
SVM formulation (linearly non-separable data)

- In SVM, we have so far assumed that data is linearly separable. What approach should we take when data is not linearly separable?
- Sometimes, data can be linearly separable but with a narrow margin.
- At other times, due to noise, some of the instances may not be linearly separable (see the figure for noisy data).

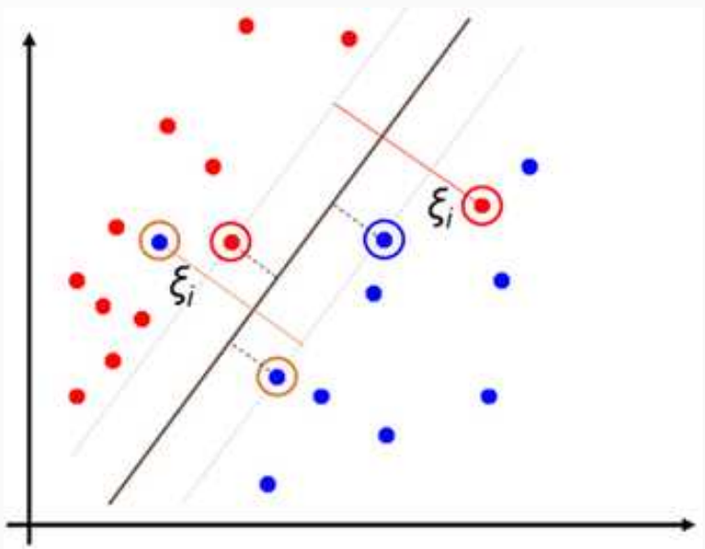


Linear SVM
SVM formulation (linearly non-separable data)

- It is generally preferred not to interfere with the boundary even with small noisy data points or outliers.
- It is acceptable to have large margins even though some of the constraints are violated.
- In practice, we need a trade-off between the margin and the number of errors in classifying the training instances.

Linear SVM
SVM formulation (linearly non-separable data)

- This trade-off brings us to the soft margin concept.
- Consider the following figure; the soft margin concept is defined when the training instances are not linearly separable.
- Slack variables are added to allow mis-classification of outliers, noisy or difficult to classify instances.
- So basically we are allowing some of the data points to cross the borders and to be in the wrong side of the boundary or to be mis-classified.



Linear SVM
SVM formulation (linearly non-separable data)

- Although we allow some of the training instances to be mis-classified, we still want to minimize the sum of slack variables.
- So for those data points which their ζ_i value is non-zero, we can infer that they are mis-classified, and the amount of mis-classification is also presented in ζ_i.
- SVM with soft margin uses the following formulation:

Find $\mathbf{w}$ and b such that

$\Phi(\mathbf{w}) = \mathbf{w}^T\mathbf{w} + C\Sigma\xi_i$ is minimized

and for all $(\mathbf{x}_i, y_i)$, $i=1..n$: $y_i(\mathbf{w}^T\mathbf{x}_i + b) \geq 1 - \xi_i$, $\xi_i \geq 0$

Linear SVM
SVM formulation (linearly non-separable data)

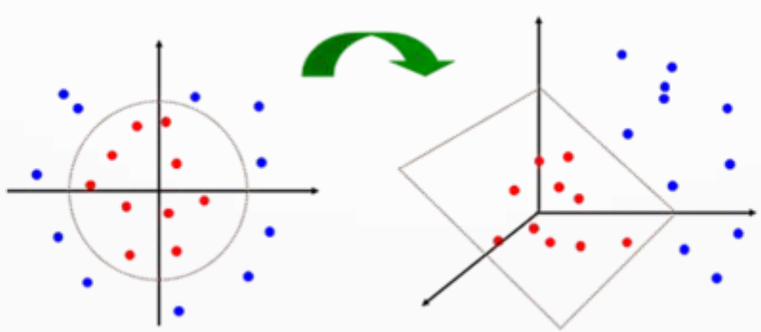
- The parameter can be used as a way to achieve the trade-off between large margin and fitting training data.
  - For the high values of , we highly penalize the mis-classification.
  - But for the small values of , we allow more mis-classifications.
- That is how SVM handles this trade-off around mis-classification.

Linear SVM
Linear SVM: Summary

- In the previous step, we investigated how an SVM handles perfectly separable data points.
- In this, we looked into how it handles almost separable data points.
- In the next part, we are going to review nonlinear SVMs.

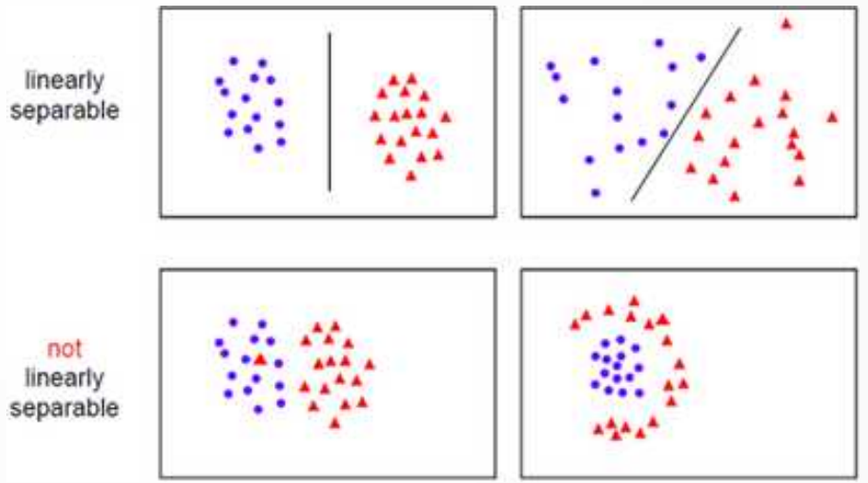
Nonlinear SVM
Kernel trick and non-linear SVM

- You may encounter data points which are not linearly separable;
- in that case, can we project them to another space in which they could be linearly separable?



Nonlinear SVM
Kernel trick and non-linear SVM...

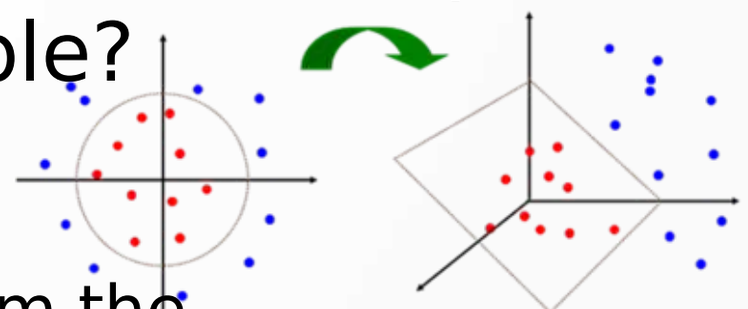
- in the first two cases, we can see that the data points are easily linearly separable.
- In the bottom left image, a single noisy data point exists; but by ignoring that points we can still draw a proper linear boundary for the data.



- since there are no linear separable lines you can understand the complexity of the classification.
- The data points and their great numbers, conclude that these are not just noisy outputs.
- So we probably need a nonlinear boundary for this.

Nonlinear SVM
Kernel trick and non-linear SVM...

- When faced with data points that are not linearly separable, can we project them to another space in which they are linearly separable?
- The key idea:
  - To handle non-linearity, we can transform the features to a higher dimensional space where data is linearly separable (see figure below).
  - This figure illustrates a 2D space in which the data points can only be separated through a nonlinear curve.
  - However by transforming these data points to a 3D space, it looks that data points are now linearly separable!



Nonlinear SVM
Kernel trick and non-linear SVM...

- SVM performs the required transformation $\Phi : \mathbf{x} \rightarrow \phi(\mathbf{x})$ implicitly by using kernels.
- We call this implicit because we do not need to compute $\phi(x)$
- Since data participates (see the previous lesson) in computations only in the form of dot products, all the computations are performed via kernels.

Nonlinear SVM
Kernel trick and non-linear SVM...

- The dot product of instances in transformed space becomes $\phi^T(\mathbf{x}_i)\phi(\mathbf{x}_j)$
- which is directly computed through a kernel function $k(\mathbf{x}_i, \mathbf{x}_j)$
- This is known as a kernel trick.
- There are kernels (e.g. radial basis function kernel) which allows transformations to even infinite dimensional feature spaces.
- A kernel function is a function that is used to compute dot products in a high dimensional feature space.

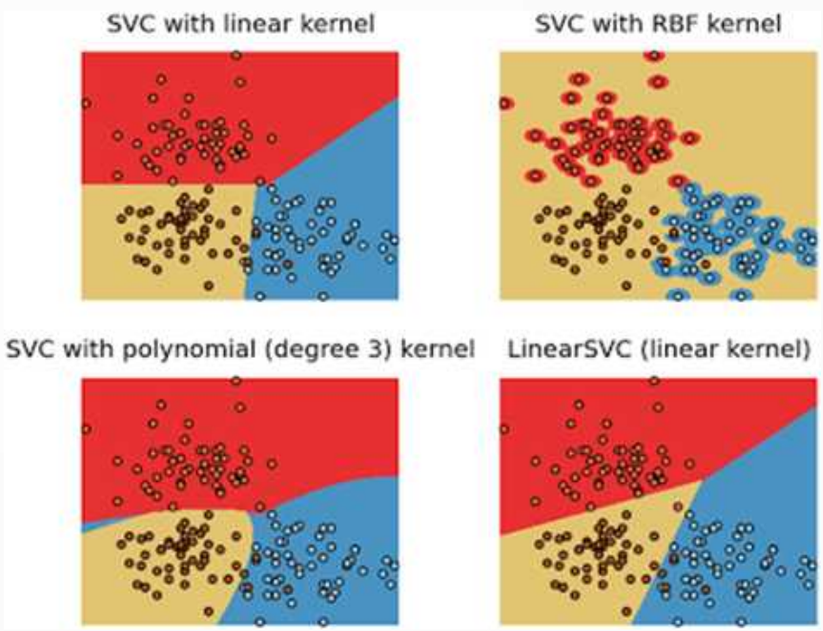
Nonlinear SVM
Kernel function...

- Some popular kernel functions where $k(\mathbf{x}_i, \mathbf{x}_j) = \phi^T(\mathbf{x}_i)\phi(\mathbf{x}_j)$
  - Linear Kernel: $k(\mathbf{x}_i, \mathbf{x}_j) = \mathbf{x}_i^T \mathbf{x}_j, Mapping \qquad \phi(\mathbf{x}) = x$
  - Polynomial Kernel with degree p: $k(\mathbf{x}_i, \mathbf{x}_j) = (1 + \mathbf{x}_i^T \mathbf{x}_j)^p$
  - Radial basis function (RBF) kernel: $k(\mathbf{x}_i, \mathbf{x}_j) = e^{-\frac{\|\mathbf{x}_i - \mathbf{x}_j\|^2}{2\sigma^2}}$

    Mapping ϕ(x) is infinite dimensional.

Nonlinear SVM
Kernel function...

- SVM fits a linear hyperplane in high dimensional space;
- but in original space the boundaries are nonlinear
- In the figure, SVC means Support Vector Classification.
- Also you can see while using the linear kernel you are getting linear boundaries,
- however in polynomial kernel with degree 3, the SVM came up with curve boundaries.



All these outputs are on the Iris data set which is a multivariate data set introduced by the British statistician and biologist Ronald Fisher.

Python Program

KNN
SVM (Linear, poly and RBF kernel)

KNN

- The K Nearest Neighbour (KNN) algorithm as one of the most interesting and powerful machine learning methods.
- In this practical, you will apply them to classification problems that are possibly non-linearly separable.
- For this practical, we use the simple but extremely popular Iris data set.
- This dataset contains 50 samples of the iris flower varieties:(Iris setosa, Iris virginica and Iris versicolor).
- Each row contains four features: the length and the width of the sepals and petals, in centimetres.
- This data is commonly use to build classification models that take these four measurements as input and predict the species of Iris flower (setosa/ virginica/versicolor).

KNN

- We can import the iris dataset from this package using the following code:

```
from matplotlib import pyplot as plt
import numpy as np

# Import the Iris data set
from sklearn import datasets
iris = datasets.load_iris()

# divide this data into features and labels
X = iris.data
y = iris.target

print('X is of type: {}'.format(type(X)))
print('y is of type: {}'.format(type(y)))

# How does our data look
print('First 5 rows of our data: {}'.format(X[:5, :]))
print('Unique labels: {}'.format(np.unique(y)))
```

```
X is of type: <class 'numpy.ndarray'>
y is of type: <class 'numpy.ndarray'>
First 5 rows of our data: [[5.1 3.5 1.4 0.2]
 [4.9 3.  1.4 0.2]
 [4.7 3.2 1.3 0.2]
 [4.6 3.1 1.5 0.2]
 [5.  3.6 1.4 0.2]]
Unique labels: [0 1 2]
```

KNN

- Let us now split our data into 80% training and 20% testing.

```
from sklearn.model_selection import train_test_split

#Split the data into 80% Training and 20% Testing sets
Xtrain, Xtest, ytrain, ytest = train_test_split(X,y, test_size=0.2, random_state=42)

print (Xtrain.shape)
print (ytrain.shape)
print (Xtest.shape)
print (ytest.shape)

Xtrain[:5,:] # first 5 rows of training data
```

```
First 5 rows of our data: [[5.1 3.5]
 [4.9 3. ]
 [4.7 3.2]
 [4.6 3.1]
 [5.  3.6]]
(120, 2)
(120,)
(30, 2)
(30,)
```

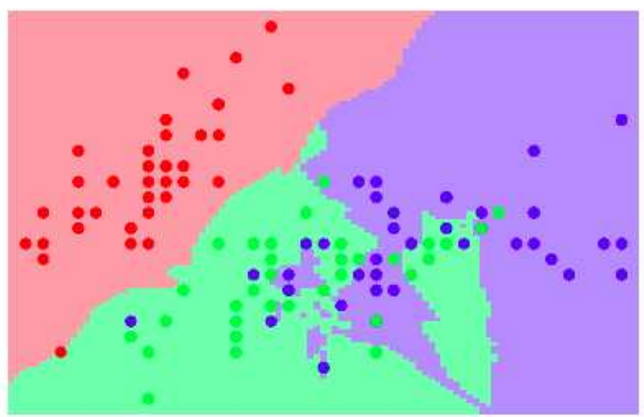
```
array([[4.6, 3.6],
       [5.7, 4.4],
       [6.7, 3.1],
       [4.8, 3.4],
       [4.4, 3.2]])
```

KNN: Advanced Visualization

- We now define a function to plot the true data points and the calculated decision boundaries of a given classifier model (also called estimator).

```
from sklearn.neighbors import KNeighborsClassifier
from sklearn import metrics

# Build a kNN using 5 neighbor nodes
knn_model = KNeighborsClassifier(n_neighbors=5)

#Fit the model using our training data
knn_model.fit(Xtrain, ytrain)

#Training Accuracy:
knn_acc = metrics.accuracy_score(ytrain, knn_model.predict(Xtrain))
print ("KNN Training Accuracy: {}".format(knn_acc))

#visualize the decision bounday. The points represent the true data.
plot_estimator(knn_model, Xtrain, ytrain)
```

```
KNN Training Accuracy: 0.8333333333333334
```



KNN: Advanced Visualization

- In this plot, the points represent the true data, the decision boundary is plotted as background color.
- We can immediately see that one class is linearly separable from the rest.
- Lets now look at the testing accuracy:

```
#Testing Accuracy:
knn_acc_test = metrics.accuracy_score(ytest, knn_model.predict(Xtest))
print ("KNN Testing Accuracy: {}".format(knn_acc_test))
```

```
KNN Testing Accuracy: 0.8
```

SVM

- Load a nonlinearly separable classification dataset
- Using the dataset, run SVM with both linear and nonlinear kernels (polynomial, RBF) and assess its prediction accuracy.
- Study the effects of C parameter and other kernel parameters and plot the performance.
- Plot the support vectors.

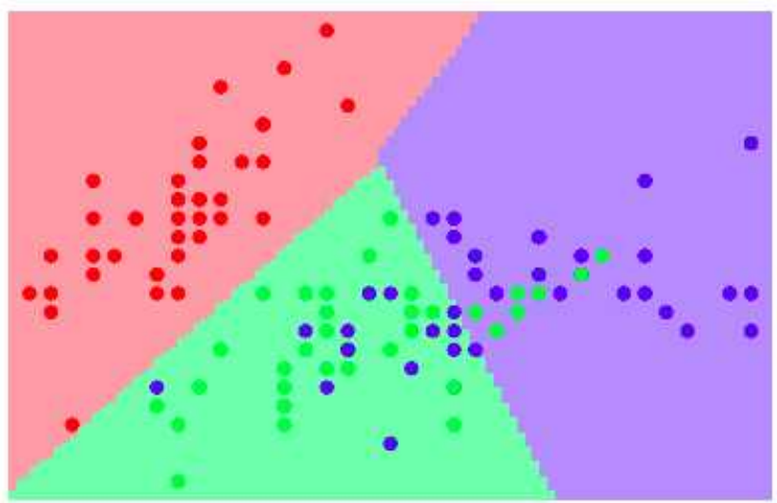
SVM...

- same as before:
  - importing libraries and dataset
  - drop last 2 columns for easy visualization
  - train-test split
- to do:
  - extend this code by building a support vector machine classification model for this data using a linear kernel.

SVM: Linear Kernel

- Let’s start with a linear kernel with the default parameters.

```
SVM Training Accuracy: 0.8
```



```
from sklearn import svm
from sklearn import metrics

# Fit an SVM using linear kernel
svm_model = svm.SVC(kernel='linear')
svm_model.fit(Xtrain, ytrain)

#Training/Testing Accuracy:
svm_acc = metrics.accuracy_score(ytrain, svm_model.predict(Xtrain))
print("SVM Training Accuracy: {}".format(svm_acc))
plot_estimator(svm_model,Xtrain,ytrain)
```

SVM: Linear Kernel

- The decision boundary is calculated using the support vectors learnt from training data.
- We can get these support vector data points using support_vectors_ parameter in the SVM model.

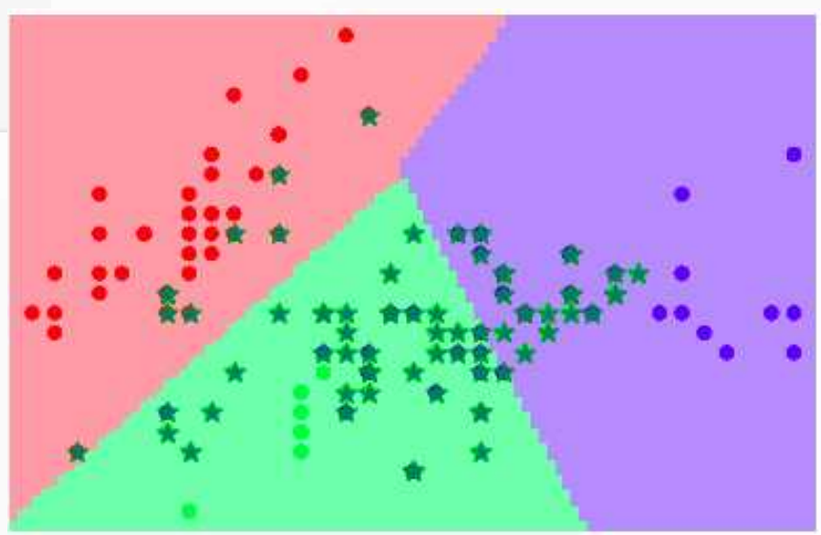
```
print(type(svm_model.support_vectors_))
print(svm_model.support_vectors_.shape)

print("Data has a total of {} support vectors"\
      .format(svm_model.support_vectors_.shape[0]))
```

```
<class 'numpy.ndarray'>
(72, 2)
Data has a total of 72 support vectors
```

SVM: Linear Kernel

- We will now plot the support vectors in * shape after plotting the decision boundary.

```
# Plotting support vectors
plot_estimator(svm_model,Xtrain,ytrain)
plt.scatter(svm_model.support_vectors_[:, 0], \
            svm_model.support_vectors_[:, 1], \
            s=100, edgecolors='g', marker='*',\
            zorder=10)
plt.show()
```

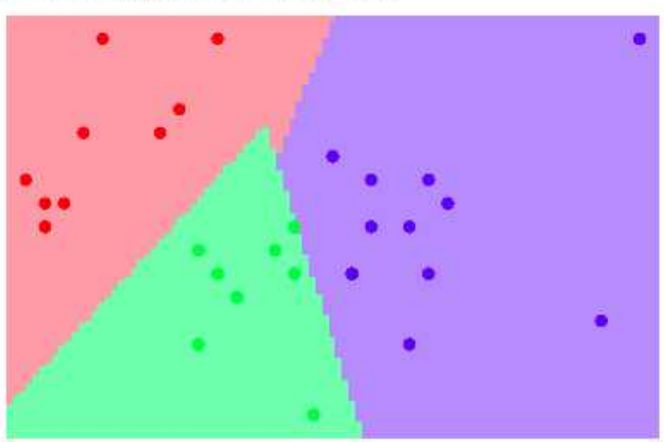


SVM: Linear Kernel

- Test accuracy.
- As we have seen Linear kernel gave us a linear decision boundary : the separation boundary is a straight line between the two categories.

```
#Testing Accuracy:
svm_acc_test = metrics.accuracy_score(ytest, \
                                      svm_model.predict(Xtest))
print("SVM Testing Accuracy: {}".format(svm_acc_test))

plot_estimator(svm_model,Xtest, svm_model.predict(Xtest))
```

```
SVM Testing Accuracy: 0.9
```



SVM: Polynomial kernel
Regularization with SVM

- In order to facilitate the visualization, let’s consider the iris data set Class 1 and Class 2 samples.
- These two types are not linearly separable , so we can see something more interesting.
- Here we use the in1d function in numpy to do this easily.

SVM: Polynomial kernel
Regularization with SVM

- In order to facilitate the visualization, let's consider the iris data set Class 1 and Class 2 samples.
- These two types are not linearly separable , so we can see something more interesting.
- Here we use the in1d function in numpy to do this easily.

```
X, y = X[np.in1d(y, [1, 2])], y[np.in1d(y, [1, 2])]

#Split the data into 80% Training and 20% Testing sets
Xtrain, Xtest, ytrain, ytest = train_test_split(X,y, \
                                                test_size=0.2,\
                                                random_state=42)

print(Xtrain.shape)
print(ytrain.shape)
print(Xtest.shape)
print(ytest.shape)
```

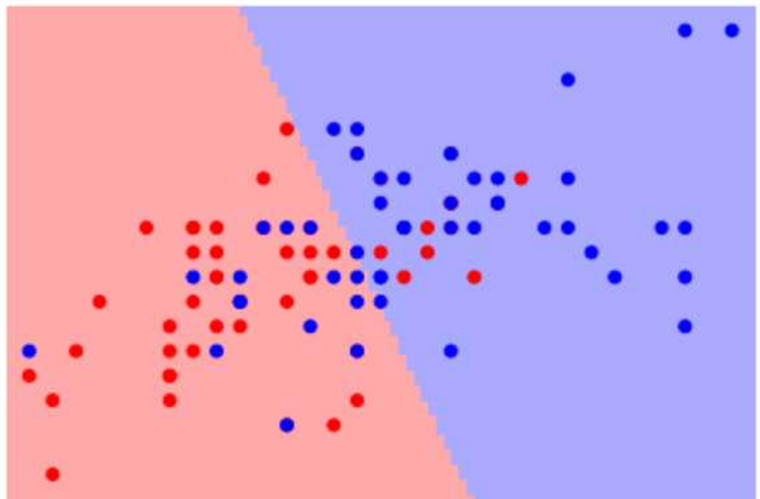
```
(80, 2)
(80,)
(20, 2)
(20,)
```

SVM: Polynomial kernel
Regularization with SVM

- Training.

```
# Fit SVM using linear kernel on training data
svc_model = svm.SVC(kernel='linear')
svc_model.fit(Xtrain, ytrain)

#Training/Testing Accuracy:
svc_acc = metrics.accuracy_score(ytrain, svc_model.predict(Xtrain))
print("SVM Training Accuracy: {}".format(svc_acc))
plot_estimator(svc_model,X,y)
```

```
SVM Training Accuracy: 0.7125
```

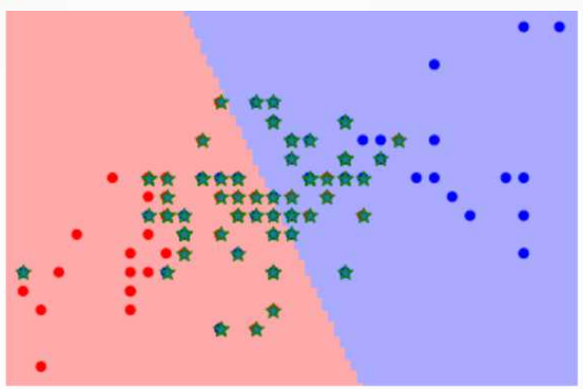


SVM: Polynomial kernel Regularization with SVM

- support vectors and testing.

```
# Plotting support vectors
plot_estimator(svc_model,X,y)
plt.scatter(svc_model.support_vectors_[:, 0], svc_model.support_vectors_[:, 1],
plt.show()
```

```
#Testing Accuracy:
svc_acc_test = metrics.accuracy_score(ytest, svc_model.predict(Xtest))
print("SVM Testing Accuracy: {}".format(svc_acc_test))
```



SVM Testing Accuracy: 0.7

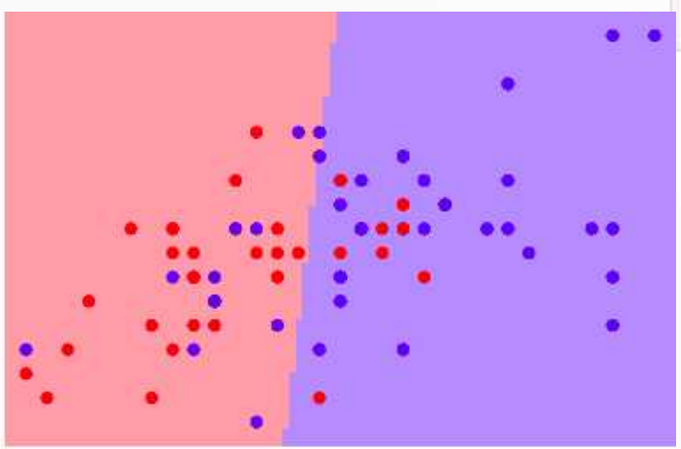
SVM: Polynomial kernel
Effect of C parameter on SVM

- C is 1 by default and it’s a reasonable default choice.
- If you have a lot of noisy observations you should decrease it.
- It corresponds to regularize more the estimation.
- It is similar to the L2 regularization parameter.
- Lets start with a high C value.
- This corresponds to high penalty for mis-classification.
- Hence the learnt margin will be narrow, resulting in small number of support vectors.

SVM: Polynomial kernel
Effect of C parameter on SVM

- code

```
svc_model = svm.SVC(kernel='linear', C=1e2)
svc_model.fit(Xtrain, ytrain)
plot_estimator(svc_model, Xtrain, ytrain)
print("Data has a total of {} support vectors"\
      .format(svc_model.support_vectors_.shape[0]))

#Training accuracy:
print("Training accuracy: {}".format(\
    metrics.accuracy_score(ytrain, svc_model.predict(Xtrain))))
print("Testing accuracy : {}".format(\
    metrics.accuracy_score(ytest, svc_model.predict(Xtest))))
```

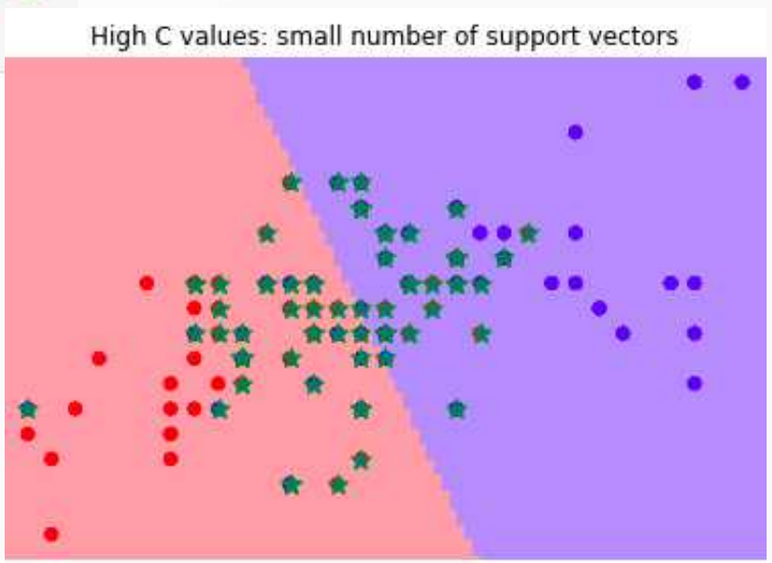


```
Data has a total of 55 support vectors
Training accuracy: 0.725
Testing accuracy : 0.7
```

SVM: Polynomial kernel
Effect of C parameter on SVM

- support vectors

```
# Plotting support vectors
plot_estimator(svc_model,X,y)
plt.scatter(svc_model.support_vectors_[:, 0],
            svc_model.support_vectors_[:, 1],
            s=100, marker='*',
            edgecolors='g',
            zorder=10)
plt.title('High C values: small number of support vectors')
plt.show()
```

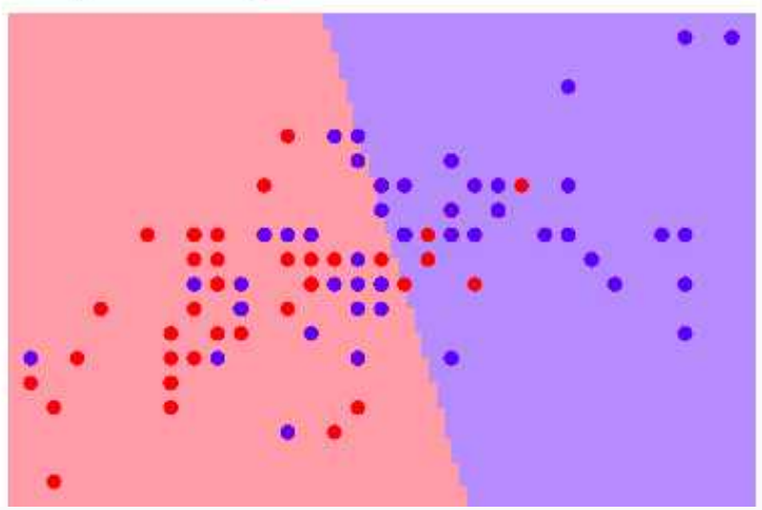


SVM: Polynomial kernel
Effect of C parameter on SVM

- If we have a low C value, we get no regularization, low penalty for misclassification.
- Hence we find a bigger margin with more support vectors.

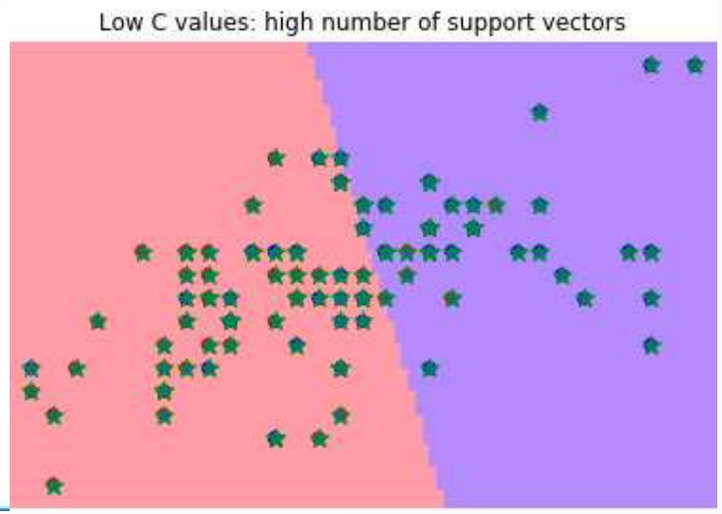
```
svc_model = svm.SVC(kernel='linear', C=1e-2)

svc_model.fit(Xtrain, ytrain)
print("Data has a total of {} support vectors"\
      .format(svc_model.support_vectors_.shape[0]))

plot_estimator(svc_model, X, y)

#Training accuracy:
print("Training accuracy: {}"\
      .format(metrics.accuracy_score(ytrain, svc_model.predict(Xtrain))))
print("Testing accuracy : {}"\
      .format(metrics.accuracy_score(ytest, svc_model.predict(Xtest))))
```

```
Data has a total of 76 support vectors
Training accuracy: 0.7
Testing accuracy : 0.65
```



SVM: Polynomial kernel
Effect of C parameter on SVM

```
# Plotting support vectors
plot_estimator(svc_model,X,y)
plt.scatter(svc_model.support_vectors_[:, 0],
            svc_model.support_vectors_[:, 1],
            s=100, marker='*',
            edgecolors='g', zorder=10)
plt.title('Low C values: high number of support vectors')
plt.show()
```

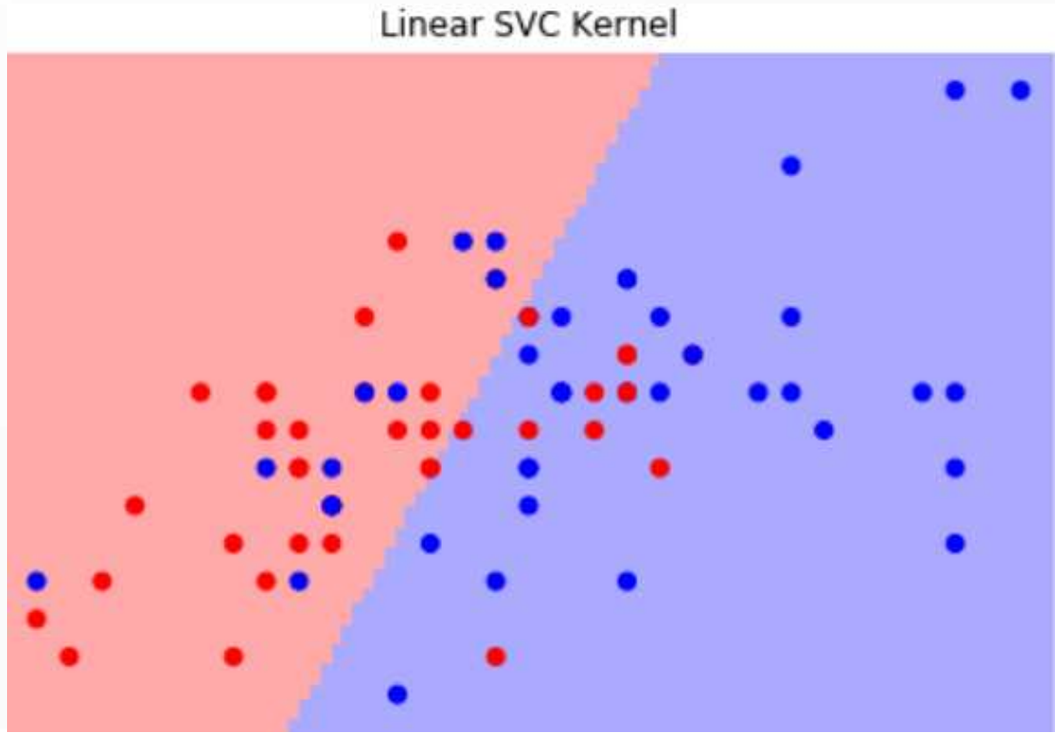


- Notice that the decision boundary has changed for the model, decreasing the testing accuracy.
- Also now almost all the points become support vectors. This is one of the issues with small sample data.

SVM: Polynomial kernel
Effect of C parameter on SVM

- In sklearn, LinearSVC is another implementation of Linear kernel.
- It has more flexibility in terms of parameters.

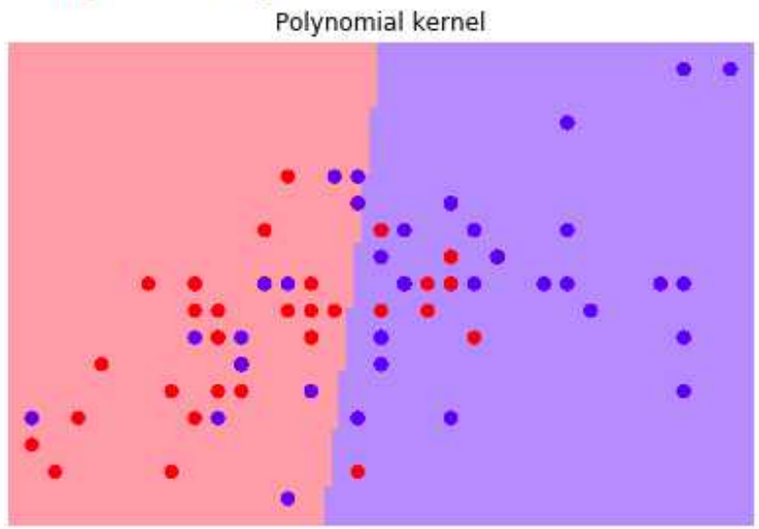
```
svc_model = svm.LinearSVC()
plot_estimator(svc_model, Xtrain, ytrain)
plt.title('Linear SVC Kernel')
plt.show()
```



SVM: Polynomial kernel
SVM with Polynomial Kernel

- Here, you fit a polynomial kernel with varying degrees.
- The degree of the polynomial kernel can be specified using “degree” parameter.

```
Training accuracy: 0.7125
Testing accuracy : 0.7
```

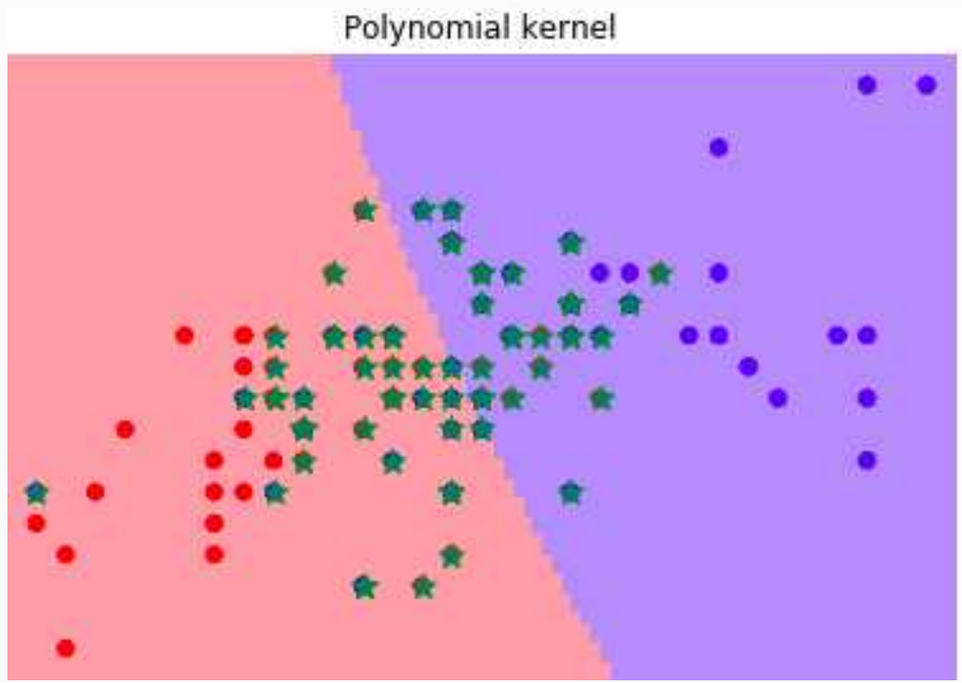



```
svc_model = svm.SVC(kernel='poly', degree=2)

svc_model.fit(Xtrain,ytrain)
plot_estimator(svc_model, Xtrain, ytrain)
plt.title('Polynomial kernel')

#Training accuracy:
print("Training accuracy: {}"\
      .format(metrics.accuracy_score(ytrain, svc_model.predict(Xtrain))))
print("Testing accuracy : {}"\
      .format(metrics.accuracy_score(ytest, svc_model.predict(Xtest))))
```

SVM: Polynomial kernel
SVM with Polynomial Kernel

- Here, you fit a polynomial kernel with varying degrees.
- The degree of the polynomial kernel can be specified using “degree” parameter.

```
#plotting support vectors
plot_estimator(svc_model, X, y)
plt.scatter(svc_model.support_vectors_[:, 0],
            svc_model.support_vectors_[:, 1],
            marker='*', s=100,
            edgecolors='g', zorder=10)
plt.title('Polynomial kernel')
plt.show()
```



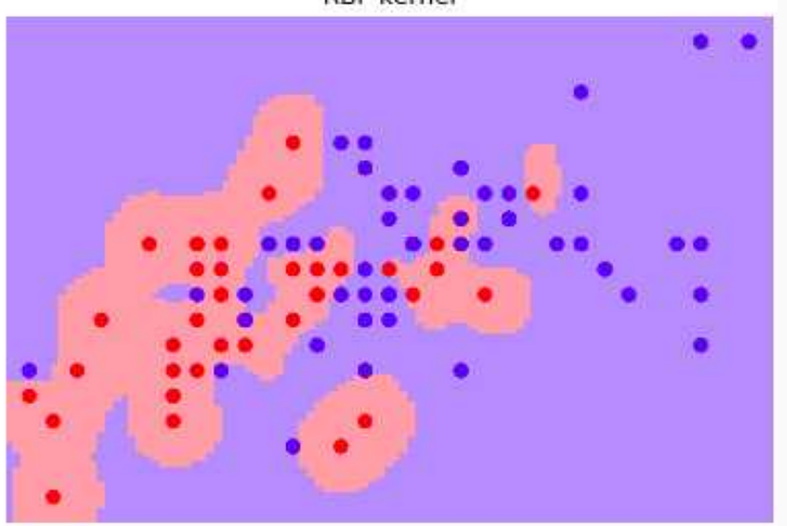
SVM: Polynomial kernel
SVM in Python - RBF kernel

- You can specify RBF Kernel using two parameters C and gamma.
- Intuitively, the gamma parameter defines how far the influence of a single training example reaches, with low values meaning ‘far’ and high values meaning ‘close’.
- The gamma parameters can be seen as the inverse of the radius of influence of samples selected by the model as support vectors.
- The C parameter trades off misclassification of training examples against simplicity of the decision surface.
- A low C makes the decision surface smooth, while a high C aims at classifying all training examples correctly by giving the model freedom to select more samples as support vectors.

SVM: Polynomial kernel
SVM in Python - RBF kernel

- For this dataset, lets look at the influence of gamma.
- You can set gamma as 10, 100, 1000 and see the differences to the decision surface and accuracy.

```
Training accuracy: 0.8875
Testing accuracy : 0.85
```




```
svc_model = svm.SVC(kernel='rbf', gamma=1e2)

svc_model.fit(X,y)
plot_estimator(svc_model, X, y)
plt.title('RBF kernel')
#Training accuracy:
print("Training accuracy: {}"\
      .format(metrics.accuracy_score(ytrain, svc_model.predict(Xtrain))))
print("Testing accuracy : {}"\
      .format(metrics.accuracy_score(ytest, svc_model.predict(Xtest))))
```

SVM: Polynomial kernel
SVM in Python - RBF kernel

- plot support vectors

```
#plotting support vectors
plot_estimator(svc_model, X, y)
plt.scatter(svc_model.support_vectors_[:, 0],
            svc_model.support_vectors_[:, 1],
            s=100, marker='*',
            edgecolors='g', zorder=10)
plt.show()
```