*You are currently looking at **version 1.1** of this notebook. To download notebooks and datafiles, as well as get help on Jupyter notebooks in the Coursera platform, visit the Jupyter Notebook FAQ (https://www.coursera.org/learn/python-machine-learning/resources/bANLa) course resource.*

Module 2: Supervised Machine Learning (Part I) ¶

02-01 : Introduction to Supervised Machine Learning

In [2]:

```
%matplotlib notebook
import numpy as np
import pandas as pd
import seaborn as sn
import matplotlib.pyplot as plt

from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier

np.set_printoptions(precision=2)


fruits = pd.read_table('readonly/fruit_data_with_colors.txt')

feature_names_fruits = ['height', 'width', 'mass', 'color_score']
X_fruits = fruits[feature_names_fruits]
y_fruits = fruits['fruit_label']
target_names_fruits = ['apple', 'mandarin', 'orange', 'lemon']

X_fruits_2d = fruits[['height', 'width']]
y_fruits_2d = fruits['fruit_label']

X_train, X_test, y_train, y_test = train_test_split(X_fruits, y_fruits, random_s
tate=0)

from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler()
X_train_scaled = scaler.fit_transform(X_train)
# we must apply the scaling to the test set that we computed for the training se
t
X_test_scaled = scaler.transform(X_test)

knn = KNeighborsClassifier(n_neighbors = 5)
knn.fit(X_train_scaled, y_train)
print('Accuracy of K-NN classifier on training set: {:.2f}'
     .format(knn.score(X_train_scaled, y_train)))
print('Accuracy of K-NN classifier on test set: {:.2f}'
     .format(knn.score(X_test_scaled, y_test)))

example_fruit = [[5.5, 2.2, 10, 0.70]]
example_fruit_scaled = scaler.transform(example_fruit)
print('Predicted fruit type for ', example_fruit, ' is ',
          target_names_fruits[knn.predict(example_fruit_scaled)[0]-1])
```

```
Accuracy of K-NN classifier on training set: 0.95
Accuracy of K-NN classifier on test set: 1.00
Predicted fruit type for  [[5.5, 2.2, 10, 0.7]]  is  mandarin
```

Classification and Regression

Both classification and regression take a set of training instances and learn a mapping to a target value.

Classification

For classification, the target value is a **discrete class value**.

- Binary: target value is 0 (negative class) or 1 (positive class) - like detecting a fraudulent (value 1) credit card transaction or a normal legitimised transaction (value 0).
- Multi Class: Target values is one of a set of discrete values - like labelling the type of fruit given data from physical attributes.
- Multi-label classification: These have **multiple target values** - like gtake a web page and classify the pages into multiple topics.

Regression

- For regression, the target value is **continuous** (floating point /real valued).
    - Like predicting the selling price of a house from its attributes

Looking at the **target value's type** will guide you onwhat supervised learning method to use (NOT the training data set). Many supervised learning methods have "flavours" for *both* classification and regression. For instance, **Support Vector** classifiers also has a regression version called "Support Vector Regression".

02-02 : Overfitting and Underfitting

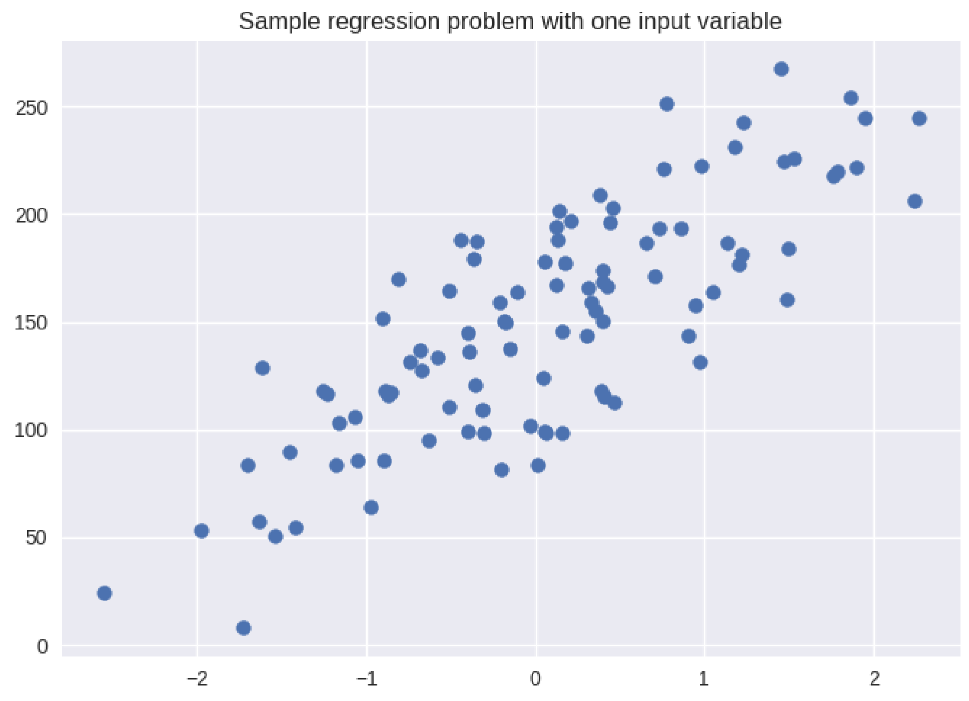
Content in slides.

02-03: Datasets

Regression

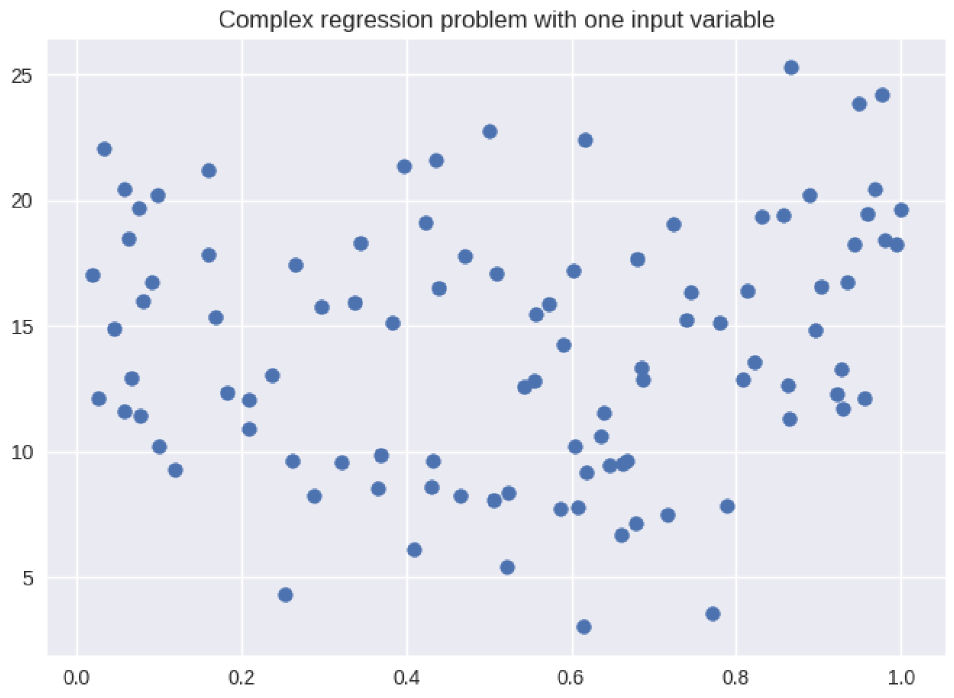
In [3]:

```
from sklearn.datasets import make_classification, make_blobs
from matplotlib.colors import ListedColormap
from sklearn.datasets import load_breast_cancer
from adspy_shared_utilities import load_crime_dataset #spoonfed!

cmap_bold = ListedColormap(['#FFFF00', '#00FF00', '#0000FF','#000000'])


# synthetic dataset for simple regression
from sklearn.datasets import make_regression ## REMEMBER THIS!
plt.figure()
plt.title('Sample regression problem with one input variable')
X_R1, y_R1 = make_regression(n_samples = 100, n_features=1,
                            n_informative=1, bias = 150.0,
                            noise = 30, random_state=0)
plt.scatter(X_R1, y_R1, marker= 'o', s=50)
plt.show()
```



In [4]:

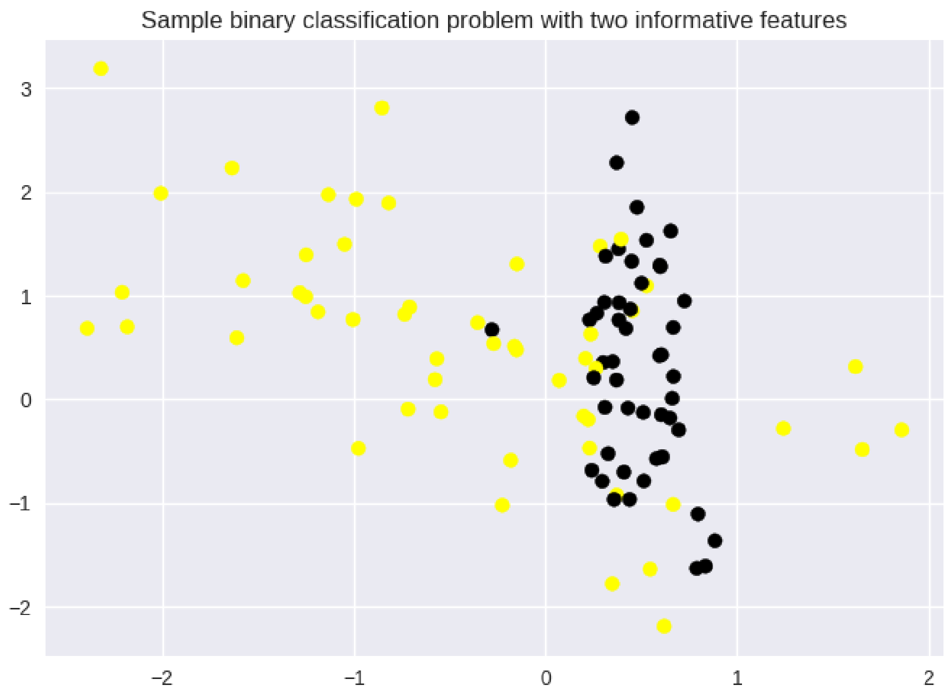
```
# synthetic dataset for more complex regression
from sklearn.datasets import make_friedman1
plt.figure()
plt.title('Complex regression problem with one input variable')
X_F1, y_F1 = make_friedman1(n_samples = 100,
                           n_features = 7, random_state=0)

plt.scatter(X_F1[:, 2], y_F1, marker= 'o', s=50)
plt.show()
```



Classification

In [5]:

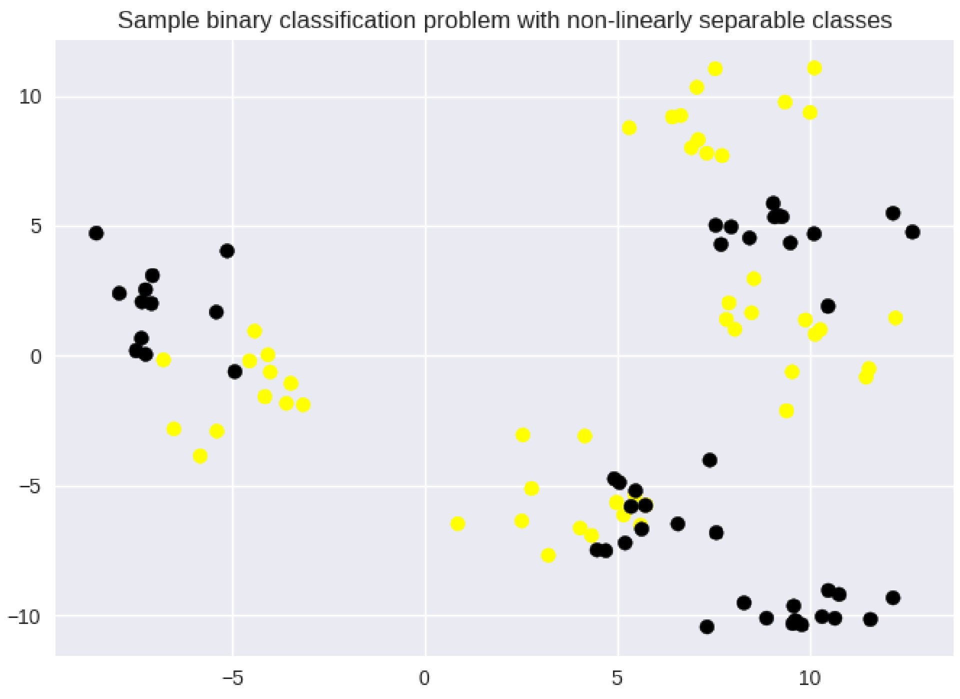
```
from sklearn.datasets import make_classification, make_blobs
from matplotlib.colors import ListedColormap
from sklearn.datasets import load_breast_cancer
from adspy_shared_utilities import load_crime_dataset

cmap_bold = ListedColormap(['#FFFF00', '#00FF00', '#0000FF','#000000'])
# synthetic dataset for classification (binary)
plt.figure()
plt.title('Sample binary classification problem with two informative features')
X_C2, y_C2 = make_classification(n_samples = 100, n_features=2,
                                n_redundant=0, n_informative=2,
                                n_clusters_per_class=1, flip_y = 0.1,
                                class_sep = 0.5, random_state=0)
plt.scatter(X_C2[:, 0], X_C2[:, 1], c=y_C2,
            marker= 'o', s=50, cmap=cmap_bold)
plt.show()
```



The below dataset was created in 2 steps. First using the `make_blobs` function in SKLearn to randomly generate 100 samples (`n_samples = 100`) in 8 different clusters (`centers = 8`). Next, we change the cluster label assigned by `make_blobs` where it's values $1 \leq$ `make_blobs()` ≤ 8, to a binary number converting it using a modulo 2 function. This assigns the even index points to class 0, and the odd index points to class 1.

In [6]:

```
# more difficult synthetic dataset for classification (binary)
# with classes that are not linearly separable
cmap_bold = ListedColormap(['#FFFF00', '#00FF00', '#0000FF','#000000'])
X_D2, y_D2 = make_blobs(n_samples = 100, n_features = 2, centers = 8,
                       cluster_std = 1.3, random_state = 4)
y_D2 = y_D2 % 2 # this converts all values in y_D2 into a binary number using th
e modulo 2 function.
plt.figure()
plt.title('Sample binary classification problem with non-linearly separable clas
ses')
plt.scatter(X_D2[:,0], X_D2[:,1], c=y_D2,
           marker= 'o', s=50, cmap=cmap_bold)
plt.show()
```



In [7]:

```
# Breast cancer dataset for classification
cancer = load_breast_cancer() #cuz we are spoon fed
(X_cancer, y_cancer) = load_breast_cancer(return_X_y = True)
X_cancer
```

Out[7]:

```
array([[  1.80e+01,   1.04e+01,   1.23e+02, ...,   2.65e-01,   4.60e
-01,
          1.19e-01],
       [  2.06e+01,   1.78e+01,   1.33e+02, ...,   1.86e-01,   2.75e
-01,
          8.90e-02],
       [  1.97e+01,   2.12e+01,   1.30e+02, ...,   2.43e-01,   3.61e
-01,
          8.76e-02],
       ...,
       [  1.66e+01,   2.81e+01,   1.08e+02, ...,   1.42e-01,   2.22e
-01,
          7.82e-02],
       [  2.06e+01,   2.93e+01,   1.40e+02, ...,   2.65e-01,   4.09e
-01,
          1.24e-01],
       [  7.76e+00,   2.45e+01,   4.79e+01, ...,   0.00e+00,   2.87e
-01,
          7.04e-02]])
```

In [8]:

```
# Communities and Crime dataset
(X_crime, y_crime) = load_crime_dataset() # cuz we are spoon fed
X_crime
```

```
Out[8]:
```

	population	householdsize	agePct12t21	agePct12t29	agePct16t24	agePct65up
0	11980	3.10	12.47	21.44	10.93	11.33
1	23123	2.82	11.01	21.30	10.48	17.18
2	29344	2.43	11.36	25.88	11.01	10.28
3	16656	2.40	12.55	25.20	12.19	17.57
5	140494	2.45	18.09	32.89	20.04	13.26
6	28700	2.60	11.17	27.41	12.76	14.42
7	59459	2.45	15.31	27.93	14.78	14.60
8	74111	2.46	16.64	35.16	20.33	8.58
9	103590	2.62	19.88	34.55	21.62	13.12
10	31601	2.54	15.73	28.57	15.16	14.26
11	25158	2.89	13.65	28.82	13.23	9.44
12	40641	2.54	21.51	36.83	23.96	11.50
13	57140	2.74	16.51	28.17	14.68	13.38
16	45532	2.85	11.86	27.51	12.36	9.76
17	180038	2.62	12.04	26.68	12.37	11.54
18	14869	2.67	13.71	20.33	9.48	12.38
19	261721	2.60	14.18	32.78	15.14	4.58
21	7322564	2.60	13.06	27.46	13.09	11.62
22	26436	3.34	16.16	37.22	19.09	4.13
23	12183	2.36	11.29	26.87	12.20	11.28
24	16023	2.63	11.91	23.09	10.33	13.65
25	24692	2.54	12.84	25.81	11.93	14.62
26	13842	2.35	8.23	18.30	6.98	13.57
27	88693	2.70	13.77	30.92	15.15	9.65
28	11235	2.60	14.59	25.62	12.02	13.36
29	28259	2.86	16.26	28.79	13.83	9.53
30	11162	2.80	15.43	26.03	12.04	12.78
31	122899	3.84	19.93	33.12	16.79	7.32
32	20714	2.36	11.35	23.13	11.19	21.15
33	11856	3.03	20.11	31.45	19.01	8.63
...	...	...	...	...	...	...
2185	48949	2.55	16.01	28.66	15.45	13.55
2186	23252	2.19	9.99	20.61	10.00	21.61

	population	householdsize	agePct12t21	agePct12t29	agePct16t24	agePct65up
2187	15066	2.40	9.98	27.58	11.45	13.34
2188	13198	2.55	10.18	22.31	9.41	13.46
2189	16829	3.17	13.76	24.66	10.40	4.26
2190	15372	3.04	13.09	21.16	10.79	12.05
2191	13333	2.69	11.55	23.33	10.16	15.85
2192	37825	2.91	13.73	23.88	12.80	13.12
2193	29885	3.53	20.10	34.33	18.31	8.18
2194	37520	2.92	14.67	27.32	12.40	7.64
2195	20319	3.02	14.78	32.22	15.93	9.05
2196	77759	2.64	23.36	38.80	26.50	10.61
2197	14904	2.50	9.83	30.35	10.27	3.03
2198	12208	2.77	13.29	25.34	13.25	16.32
2199	25430	3.03	13.47	25.27	11.47	5.91
2200	92703	2.49	13.45	27.70	13.93	16.06
2201	24544	2.71	14.33	26.72	12.34	8.78
2202	14203	2.97	15.13	27.38	12.31	10.13
2203	27331	2.96	13.99	33.75	15.53	5.34
2204	23875	2.67	13.62	25.83	11.67	10.84
2205	14804	2.60	14.37	25.24	12.75	15.16
2206	21732	2.42	12.90	22.30	10.07	17.83
2207	54021	2.54	11.86	27.72	13.18	13.39
2208	13131	2.56	13.81	24.03	10.78	12.99
2209	10567	2.57	10.42	24.90	11.10	15.33
2210	56216	3.07	15.46	30.16	14.34	8.08
2211	12251	2.68	17.36	31.23	16.97	12.57
2212	32824	2.46	11.81	20.96	9.53	20.73
2213	13547	2.89	17.16	30.01	14.73	10.42
2214	28898	2.61	12.99	25.21	11.63	12.12

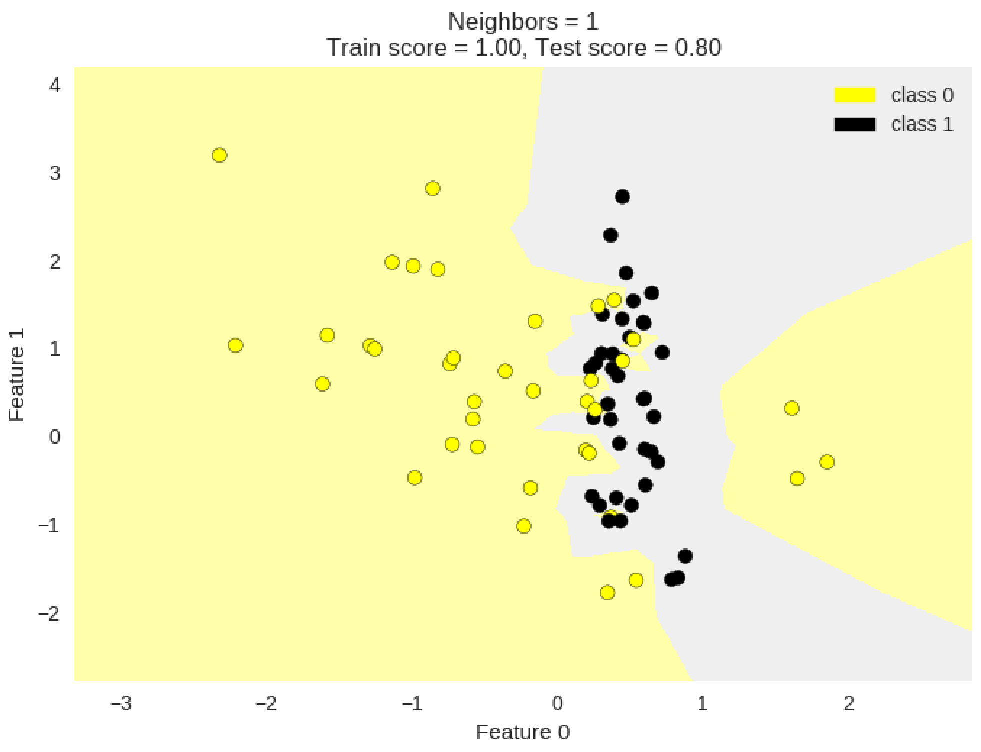
1994 rows × 88 columns

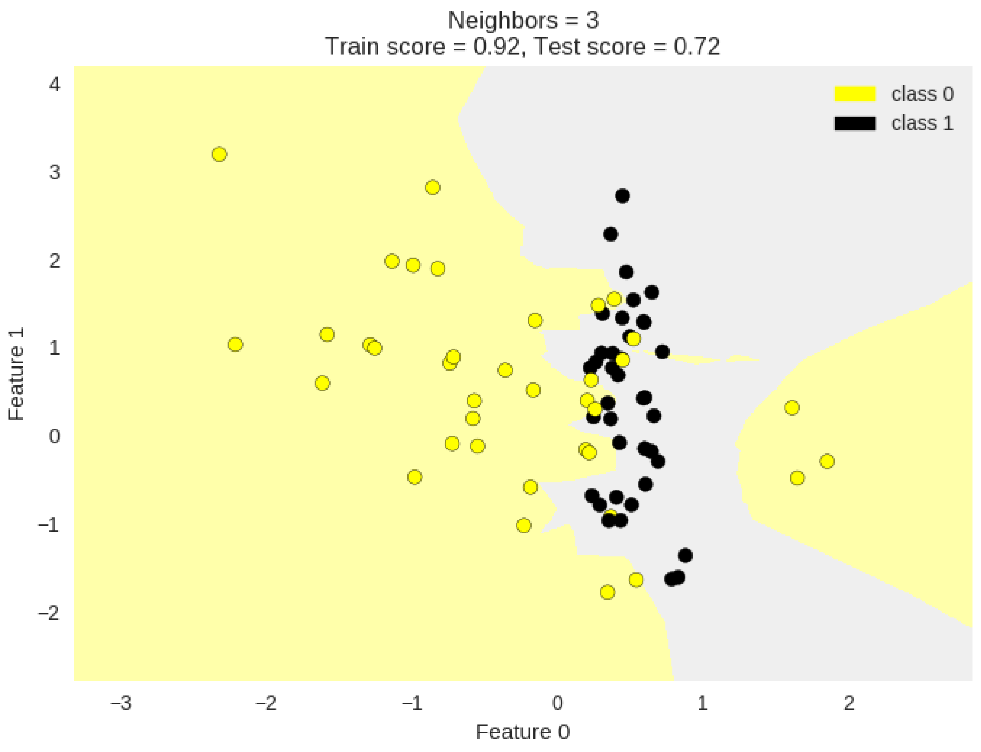
02-04: K-Nearest Neighbors - Classification & Regression

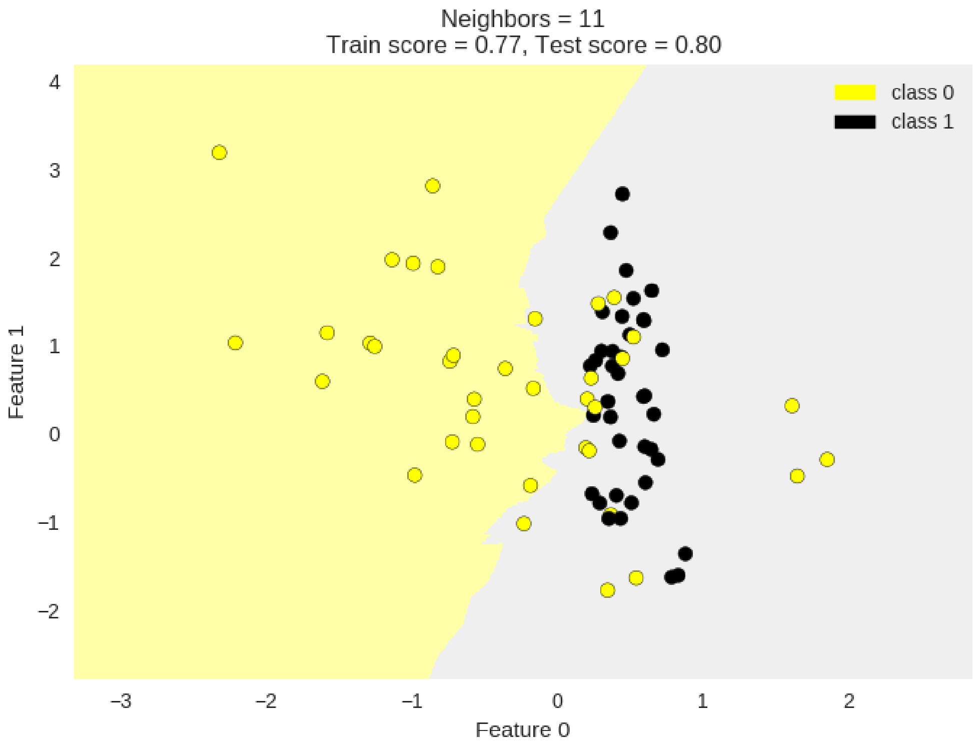
Classification

In [9]:

```
from adspy_shared_utilities import plot_two_class_knn #spoon fed again!

X_train, X_test, y_train, y_test = train_test_split(X_C2, y_C2,
                                                    random_state=0)

plot_two_class_knn(X_train, y_train, 1, 'uniform', X_test, y_test)
plot_two_class_knn(X_train, y_train, 3, 'uniform', X_test, y_test)
plot_two_class_knn(X_train, y_train, 11, 'uniform', X_test, y_test)
```


Neighbors = 1
Train score = 1.00, Test score = 0.80
class 0
class 1
Feature 0
Feature 1
-3
-2
-1
0
1
2
3
4


Neighbors = 3
Train score = 0.92, Test score = 0.72
class 0
class 1
Feature 0
Feature 1
-3
-2
-1
0
1
2
3
4



Regression

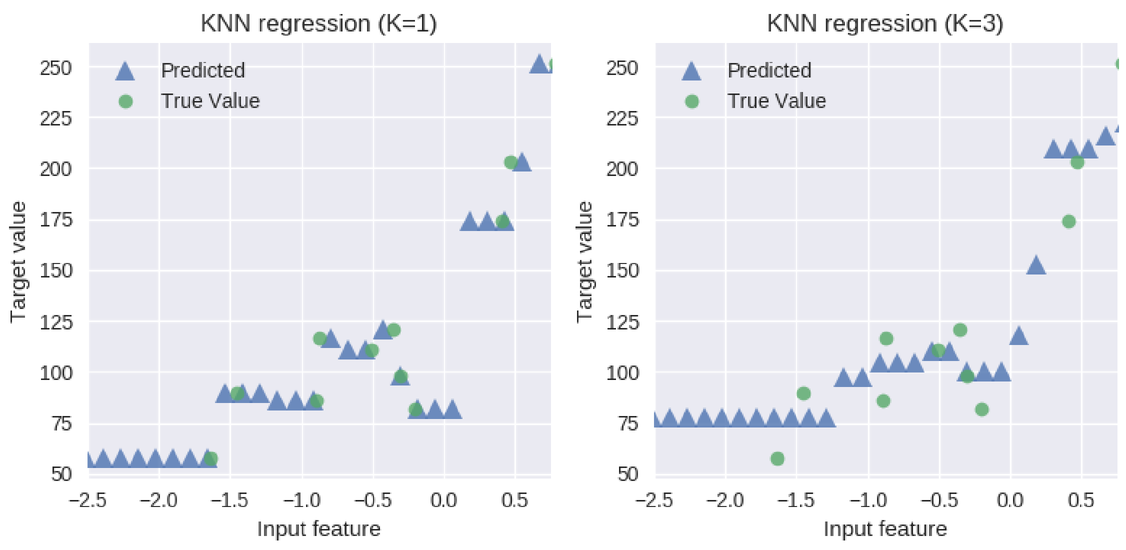
In [10]:

```
from sklearn.neighbors import KNeighborsRegressor

X_train, X_test, y_train, y_test = train_test_split(X_R1, y_R1, random_state = 0
)

knnreg = KNeighborsRegressor(n_neighbors = 5).fit(X_train, y_train)

print(knnreg.predict(X_test))
print('R-squared test score: {:.3f}'
     .format(knnreg.score(X_test, y_test)))
```

```
[ 231.71   148.36   150.59   150.59    72.15   166.51   141.91   235.57   20
8.26
  102.1    191.32   134.5    228.32   148.36   159.17   113.47   144.04   19
9.23
  143.19   166.51   231.71   208.26   128.02   123.14   141.91]
R-squared test score: 0.425
```

In [11]:

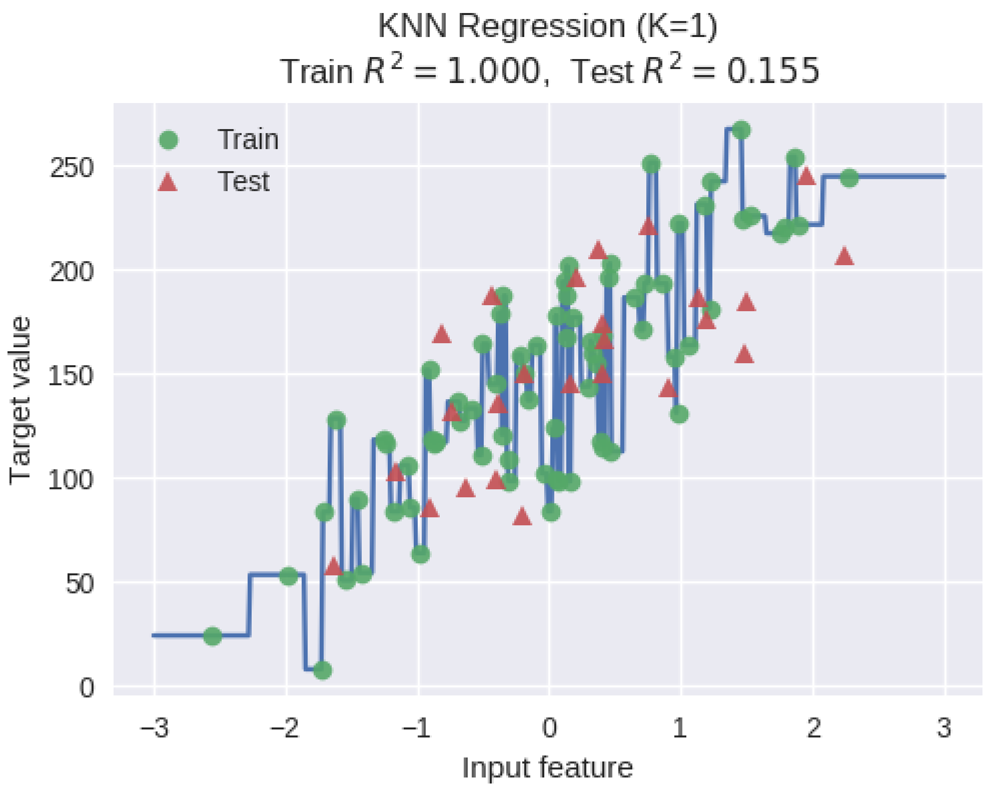
```
fig, subaxes = plt.subplots(1, 2, figsize=(8,4))
X_predict_input = np.linspace(-3, 3, 50).reshape(-1,1)
X_train, X_test, y_train, y_test = train_test_split(X_R1[0::5], y_R1[0::5], rand
om_state = 0)

for thisaxis, K in zip(subaxes, [1, 3]):
    knnreg = KNeighborsRegressor(n_neighbors = K).fit(X_train, y_train)
    y_predict_output = knnreg.predict(X_predict_input)
    thisaxis.set_xlim([-2.5, 0.75])
    thisaxis.plot(X_predict_input, y_predict_output, '^', markersize = 10,
                 label='Predicted', alpha=0.8)
    thisaxis.plot(X_train, y_train, 'o', label='True Value', alpha=0.8)
    thisaxis.set_xlabel('Input feature')
    thisaxis.set_ylabel('Target value')
    thisaxis.set_title('KNN regression (K={})'.format(K))
    thisaxis.legend()
plt.tight_layout()
```

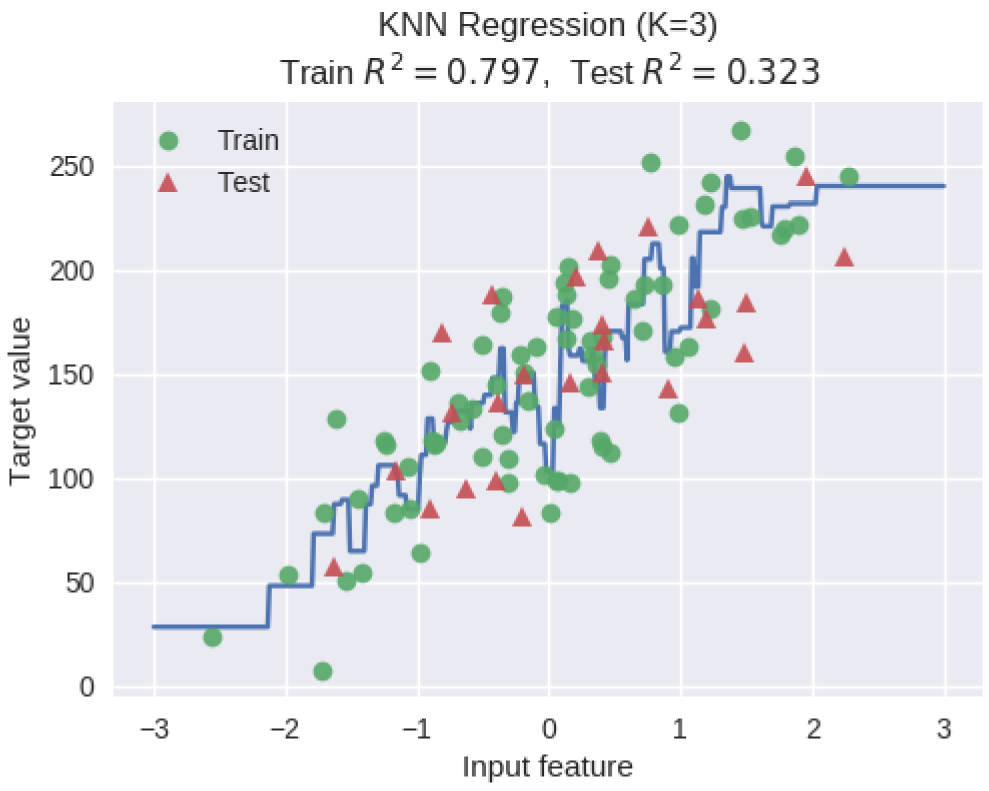


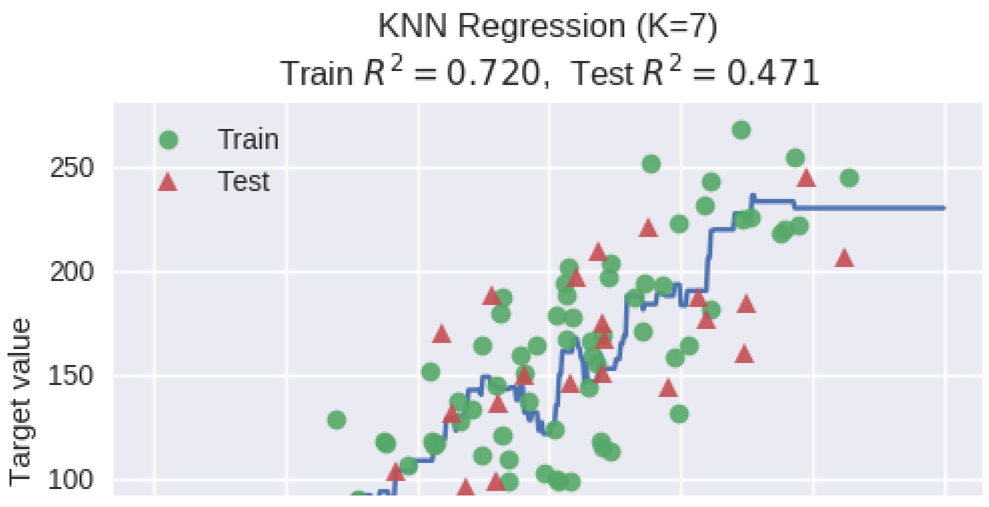
Regression model complexity as a function of K

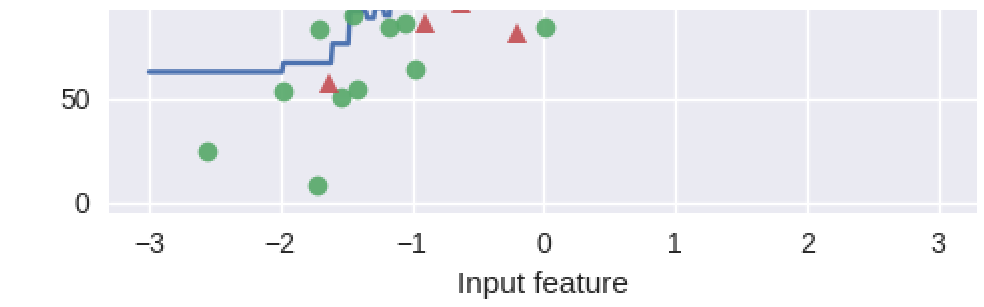
In [12]:

```
# plot k-NN regression on sample dataset for different values of K
fig, subaxes = plt.subplots(5, 1, figsize=(5,20))
X_predict_input = np.linspace(-3, 3, 500).reshape(-1,1)
X_train, X_test, y_train, y_test = train_test_split(X_R1, y_R1,
                                                    random_state = 0)

for thisaxis, K in zip(subaxes, [1, 3, 7, 15, 55]):
    knnreg = KNeighborsRegressor(n_neighbors = K).fit(X_train, y_train)
    y_predict_output = knnreg.predict(X_predict_input)
    train_score = knnreg.score(X_train, y_train)
    test_score = knnreg.score(X_test, y_test)
    thisaxis.plot(X_predict_input, y_predict_output)
    thisaxis.plot(X_train, y_train, 'o', alpha=0.9, label='Train')
    thisaxis.plot(X_test, y_test, '^', alpha=0.9, label='Test')
    thisaxis.set_xlabel('Input feature')
    thisaxis.set_ylabel('Target value')
    thisaxis.set_title('KNN Regression (K={})\n\
Train $R^2 = {:.3f}$,  Test $R^2 = {:.3f}$'
                      .format(K, train_score, test_score))
    thisaxis.legend()
    plt.tight_layout(pad=0.4, w_pad=0.5, h_pad=1.0)
```

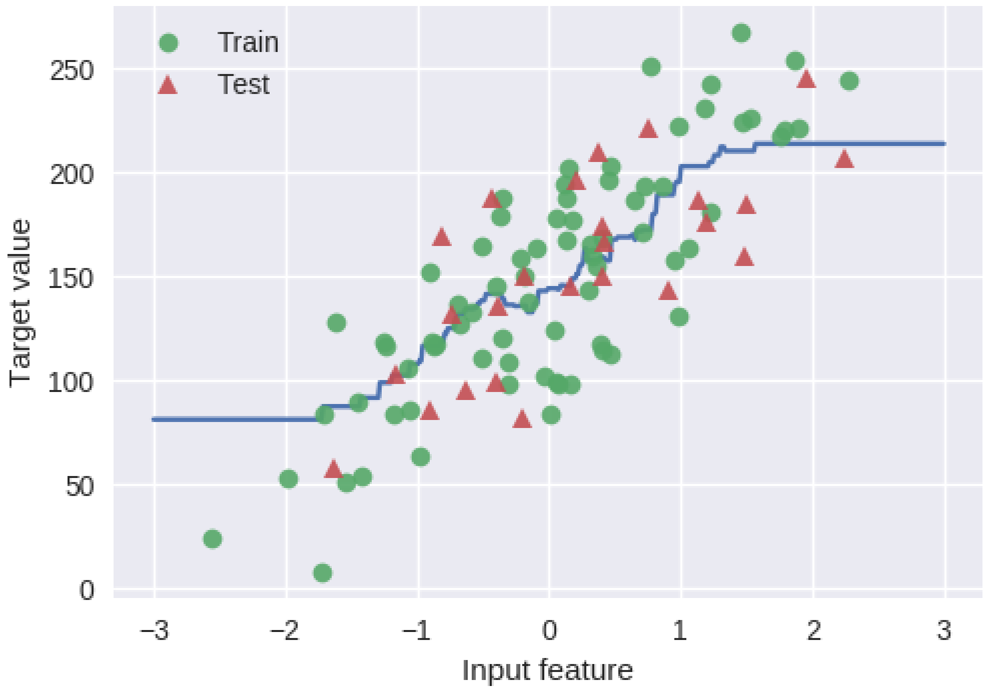

KNN Regression (K=1)
Train $R^2 = 1.000$, Test $R^2 = 0.155$
Train
Test
Target value
Input feature


KNN Regression (K=3)
Train $R^2 = 0.797$, Test $R^2 = 0.323$
Train
Test
Target value
Input feature

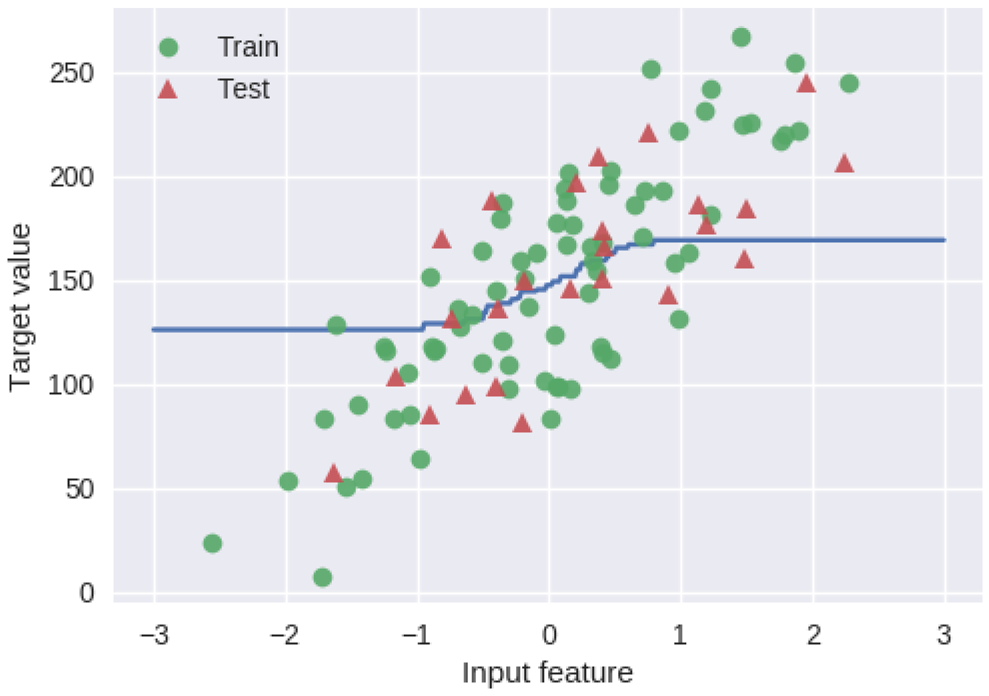

KNN Regression (K=7)
Train $R^2 = 0.720$, Test $R^2 = 0.471$
Train
Test
Target value



KNN Regression (K=15)
Train $R^2 = 0.647$, Test $R^2 = 0.485$



KNN Regression (K=55)
Train $R^2 = 0.357$, Test $R^2 = 0.371$



02-05: Linear Regression: Least Squares

Linear regression

```
In [13]:
```

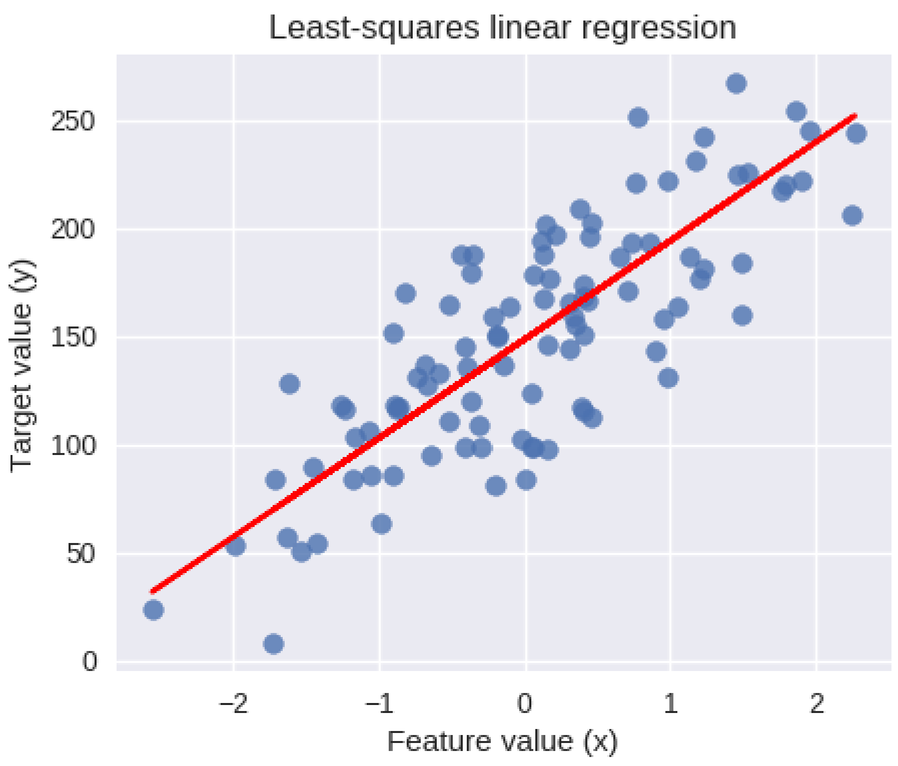
```
from sklearn.linear_model import LinearRegression

X_train, X_test, y_train, y_test = train_test_split(X_R1, y_R1,
                                                   random_state = 0) #we use the
train_test split
linreg = LinearRegression().fit(X_train, y_train) # call the constructor for lin
ear regression and use the fit method
# on this constructor, passing parameters x_train and y_train.
# this linear regression fit method acts to estimate the feature weights w, whic
h are called the coefficients
# of the model, accessible using the coef_ attrtibute.
# the bias term b is the intercept of the model and is accessible from the inter
cept_ attribute of the model.

# Note that the underscore after an attribute means that these values were deriv
ed from the training data, and not
# quantities that were set by the user.
print('linear model coeff (w): {}'
     .format(linreg.coef_))
print('linear model intercept (b): {:.3f}'
     .format(linreg.intercept_))
print('R-squared score (training): {:.3f}'
     .format(linreg.score(X_train, y_train)))
print('R-squared score (test): {:.3f}'
     .format(linreg.score(X_test, y_test)))
```

```
linear model coeff (w): [ 45.71]
linear model intercept (b): 148.446
R-squared score (training): 0.679
R-squared score (test): 0.492
```

Linear regression: example plot

In [14]:

```
plt.figure(figsize=(5,4))
plt.scatter(X_R1, y_R1, marker= 'o', s=50, alpha=0.8)
plt.plot(X_R1, linreg.coef_ * X_R1 + linreg.intercept_, 'r-')
plt.title('Least-squares linear regression')
plt.xlabel('Feature value (x)')
plt.ylabel('Target value (y)')
plt.show()
```



Using input variables of higher dimensions

In [15]:

```
X_train, X_test, y_train, y_test = train_test_split(X_crime, y_crime,
                                                    random_state = 0)
linreg = LinearRegression().fit(X_train, y_train)

print('Crime dataset')
print('linear model intercept: {}'
     .format(linreg.intercept_))
print('linear model coeff:\n{}'
     .format(linreg.coef_))
print('R-squared score (training): {:.3f}'
     .format(linreg.score(X_train, y_train)))
print('R-squared score (test): {:.3f}'
     .format(linreg.score(X_test, y_test)))
```

```
Crime dataset
linear model intercept: -1728.1306726048451
linear model coeff:
[  1.62e-03  -9.43e+01   1.36e+01  -3.13e+01  -8.15e-02  -1.69e+01
  -2.43e-03   1.53e+00  -1.39e-02  -7.72e+00   2.28e+01  -5.66e+00
   9.35e+00   2.07e-01  -7.43e+00   9.66e-03   4.38e-03   4.80e-03
  -4.46e+00  -1.61e+01   8.83e+00  -5.07e-01  -1.42e+00   8.18e+00
  -3.87e+00  -3.54e+00   4.49e+00   9.31e+00   1.74e+02   1.18e+01
   1.51e+02  -3.30e+02  -1.35e+02   6.95e-01  -2.38e+01   2.77e+00
   3.82e-01   4.39e+00  -1.06e+01  -4.92e-03   4.14e+01  -1.16e-03
   1.19e+00   1.75e+00  -3.68e+00   1.60e+00  -8.42e+00  -3.80e+01
   4.74e+01  -2.51e+01  -2.88e-01  -3.66e+01   1.90e+01  -4.53e+01
   6.83e+02   1.04e+02  -3.29e+02  -3.14e+01   2.74e+01   5.12e+00
   6.92e+01   1.98e-02  -6.12e-01   2.65e+01   1.01e+01  -1.59e+00
   2.24e+00   7.38e+00  -3.14e+01  -9.78e-05   5.02e-05  -3.48e-04
  -2.50e-04  -5.27e-01  -5.17e-01  -4.10e-01   1.16e-01   1.46e+00
  -3.04e-01   2.44e+00  -3.66e+01   1.41e-01   2.89e-01   1.77e+01
   5.97e-01   1.98e+00  -1.36e-01  -1.85e+00]
R-squared score (training): 0.673
R-squared score (test): 0.496
```

02-06: Linear Regression: Ridge, Lasso and Polynomial Regression

Ridge regression

In [16]:

```
from sklearn.linear_model import Ridge
X_train, X_test, y_train, y_test = train_test_split(X_crime, y_crime,
                                                   random_state = 0)

linridge = Ridge(alpha=20.0).fit(X_train, y_train)

print('Crime dataset')
print('ridge regression linear model intercept: {}'
     .format(linridge.intercept_))
print('ridge regression linear model coeff:\n{}'
     .format(linridge.coef_))
print('R-squared score (training): {:.3f}'
     .format(linridge.score(X_train, y_train)))
print('R-squared score (test): {:.3f}'
     .format(linridge.score(X_test, y_test)))
print('Number of non-zero features: {}'
     .format(np.sum(linridge.coef_ != 0)))
```

```
Crime dataset
ridge regression linear model intercept: -3352.423035846206
ridge regression linear model coeff:
[  1.95e-03   2.19e+01   9.56e+00  -3.59e+01   6.36e+00  -1.97e+01
  -2.81e-03   1.66e+00  -6.61e-03  -6.95e+00   1.72e+01  -5.63e+00
   8.84e+00   6.79e-01  -7.34e+00   6.70e-03   9.79e-04   5.01e-03
  -4.90e+00  -1.79e+01   9.18e+00  -1.24e+00   1.22e+00   1.03e+01
  -3.78e+00  -3.73e+00   4.75e+00   8.43e+00   3.09e+01   1.19e+01
  -2.05e+00  -3.82e+01   1.85e+01   1.53e+00  -2.20e+01   2.46e+00
   3.29e-01   4.02e+00  -1.13e+01  -4.70e-03   4.27e+01  -1.23e-03
   1.41e+00   9.35e-01  -3.00e+00   1.12e+00  -1.82e+01  -1.55e+01
   2.42e+01  -1.32e+01  -4.20e-01  -3.60e+01   1.30e+01  -2.81e+01
   4.39e+01   3.87e+01  -6.46e+01  -1.64e+01   2.90e+01   4.15e+00
   5.34e+01   1.99e-02  -5.47e-01   1.24e+01   1.04e+01  -1.57e+00
   3.16e+00   8.78e+00  -2.95e+01  -2.33e-04   3.14e-04  -4.14e-04
  -1.80e-04  -5.74e-01  -5.18e-01  -4.21e-01   1.53e-01   1.33e+00
   3.85e+00   3.03e+00  -3.78e+01   1.38e-01   3.08e-01   1.57e+01
   3.31e-01   3.36e+00   1.61e-01  -2.68e+00]
R-squared score (training): 0.671
R-squared score (test): 0.494
Number of non-zero features: 88
```

Ridge regression with feature normalization

In [17]:

```
from sklearn.preprocessing import MinMaxScaler # import the MinMaxScalar object
 from sklearn.preprocessing
scaler = MinMaxScaler() # constructor

from sklearn.linear_model import Ridge
X_train, X_test, y_train, y_test = train_test_split(X_crime, y_crime,
                                                   random_state = 0)

X_train_scaled = scaler.fit_transform(X_train) #scales all the input features on
the training set
#this computes the min and max feature values for each feature in this training
 dataset.
# to apply the scalar you call its transform() method and pass in the data you w
ant to rescale.
# The output will be a scaled version of the input data.

X_test_scaled = scaler.transform(X_test) #scales all the input features on the t
est set
# Use X_train_scaled and X_test_scaled instead of the original values in the dat
aset.
linridge = Ridge(alpha=20.0).fit(X_train_scaled, y_train)

print('Crime dataset')
print('ridge regression linear model intercept: {}'
     .format(linridge.intercept_))
print('ridge regression linear model coeff:\n{}'
     .format(linridge.coef_))
print('R-squared score (training): {:.3f}'
     .format(linridge.score(X_train_scaled, y_train)))
print('R-squared score (test): {:.3f}'
     .format(linridge.score(X_test_scaled, y_test)))
print('Number of non-zero features: {}'
     .format(np.sum(linridge.coef_ != 0)))
```

```
Crime dataset
ridge regression linear model intercept: 933.390638504416
ridge regression linear model coeff:
[  88.69   16.49  -50.3   -82.91  -65.9    -2.28   87.74  150.95   1
8.88
  -31.06  -43.14 -189.44   -4.53  107.98  -76.53    2.86   34.95   9
0.14
   52.46  -62.11  115.02    2.67    6.94   -5.67 -101.55  -36.91   -
8.71
   29.12  171.26   99.37   75.07  123.64   95.24 -330.61 -442.3  -28
4.5
 -258.37   17.66 -101.71  110.65  523.14   24.82    4.87  -30.47   -
3.52
   50.58   10.85   18.28   44.11   58.34   67.09  -57.94  116.14   5
3.81
   49.02   -7.62   55.14  -52.09  123.39   77.13   45.5   184.91  -9
1.36
    1.08  234.09   10.39   94.72  167.92  -25.14   -1.18   14.6    3
6.77
   53.2   -78.86   -5.9    26.05  115.15   68.74   68.29   16.53  -9
7.91
  205.2    75.97   61.38  -79.83   67.27   95.67  -11.88]
R-squared score (training): 0.615
R-squared score (test): 0.599
Number of non-zero features: 88
```

Ridge regression with regularization parameter: alpha

The best R^2 value on the test set is achieved with an $\alpha \approx 20$. Significantly larger of smaller values of α both leads to significantly worse model fit. This is another illustration of the general relationship with model complexity and test set performance that we saw earlier in this lecture - where there is always some **intermediate best value** of a model complexity parameter that **does not lead to underfitting or overfitting**.

In [18]:

```
print('Ridge regression: effect of alpha regularization parameter\n')
for this_alpha in [0, 1, 10, 20, 50, 100, 1000]:
    linridge = Ridge(alpha = this_alpha).fit(X_train_scaled, y_train)
    r2_train = linridge.score(X_train_scaled, y_train)
    r2_test = linridge.score(X_test_scaled, y_test)
    num_coeff_bigger = np.sum(abs(linridge.coef_) > 1.0)
    print('Alpha = {:.2f}\nnum abs(coeff) > 1.0: {}, \
r-squared training: {:.2f}, r-squared test: {:.2f}\n'
         .format(this_alpha, num_coeff_bigger, r2_train, r2_test))
```

```
Ridge regression: effect of alpha regularization parameter


/opt/conda/lib/python3.6/site-packages/scipy/linalg/basic.py:223: Ru
ntimeWarning: scipy.linalg.solve
Ill-conditioned matrix detected. Result is not guaranteed to be accu
rate.
Reciprocal condition number: 6.332952875642905e-19
  ' condition number: {}'.format(rcond), RuntimeWarning)

Alpha = 0.00
num abs(coeff) > 1.0: 88, r-squared training: 0.67, r-squared test:
0.50

Alpha = 1.00
num abs(coeff) > 1.0: 87, r-squared training: 0.66, r-squared test:
0.56

Alpha = 10.00
num abs(coeff) > 1.0: 87, r-squared training: 0.63, r-squared test:
0.59

Alpha = 20.00
num abs(coeff) > 1.0: 88, r-squared training: 0.61, r-squared test:
0.60

Alpha = 50.00
num abs(coeff) > 1.0: 86, r-squared training: 0.58, r-squared test:
0.58

Alpha = 100.00
num abs(coeff) > 1.0: 87, r-squared training: 0.55, r-squared test:
0.55

Alpha = 1000.00
num abs(coeff) > 1.0: 84, r-squared training: 0.31, r-squared test:
0.30
```

Lasso regression

In [19]:

```
from sklearn.linear_model import Lasso
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler() # constructor for feature normalisation.

X_train, X_test, y_train, y_test = train_test_split(X_crime, y_crime,
                                                   random_state = 0)

X_train_scaled = scaler.fit_transform(X_train) #feature normalise
X_test_scaled = scaler.transform(X_test) # feature normalise

linlasso = Lasso(alpha=2.0, max_iter = 10000).fit(X_train_scaled, y_train)
## In some cases you may get a convergence warning, and what you can do is to se
t the max_iter attribute to a larger
## value. Increasing the max_iter parameter will increase the computation time a
ccordingly.
print('Crime dataset')
print('lasso regression linear model intercept: {}'
     .format(linlasso.intercept_))
print('lasso regression linear model coeff:\n{}'
     .format(linlasso.coef_))
print('Non-zero features: {}'
     .format(np.sum(linlasso.coef_ != 0)))
print('R-squared score (training): {:.3f}'
     .format(linlasso.score(X_train_scaled, y_train)))
print('R-squared score (test): {:.3f}\n'
     .format(linlasso.score(X_test_scaled, y_test)))
print('Features with non-zero weight (sorted by absolute magnitude):')

for e in sorted (list(zip(list(X_crime), linlasso.coef_)),
                key = lambda e: -abs(e[1])):
    if e[1] != 0:
        print('\t{}, {:.3f}'.format(e[0], e[1]))
```

```
Crime dataset
lasso regression linear model intercept: 1186.6120619985809
lasso regression linear model coeff:
[    0.       0.      -0.    -168.18    -0.      -0.       0.     11
9.69
     0.      -0.       0.    -169.68    -0.       0.      -0.
0.
     0.       0.      -0.      -0.       0.      -0.       0.
0.
   -57.53    -0.      -0.       0.     259.33    -0.       0.
0.
     0.      -0.   -1188.74    -0.      -0.      -0.    -231.42
0.
  1488.37     0.      -0.      -0.      -0.       0.       0.
0.
     0.       0.      -0.       0.      20.14     0.       0.
0.
     0.       0.     339.04     0.       0.     459.54    -0.
0.
   122.69    -0.      91.41     0.      -0.       0.       0.      7
3.14
     0.      -0.       0.       0.      86.36     0.       0.
0.
  -104.57   264.93     0.      23.45   -49.39     0.       5.2
0.  ]
Non-zero features: 20
R-squared score (training): 0.631
R-squared score (test): 0.624

Features with non-zero weight (sorted by absolute magnitude):
        PctKidsBornNeverMar, 1488.365
        PctKids2Par, -1188.740
        HousVacant, 459.538
        PctPersDenseHous, 339.045
        NumInShelters, 264.932
        MalePctDivorce, 259.329
        PctWorkMom, -231.423
        pctWInvInc, -169.676
        agePct12t29, -168.183
        PctVacantBoarded, 122.692
        pctUrban, 119.694
        MedOwnCostPctIncNoMtg, -104.571
        MedYrHousBuilt, 91.412
        RentQrange, 86.356
        OwnOccHiQuart, 73.144
        PctEmplManu, -57.530
        PctBornSameState, -49.394
        PctForeignBorn, 23.449
        PctLargHouseFam, 20.144
        PctSameCity85, 5.198
```

Lasso regression with regularization parameter: alpha

Just like ridge regression, we can see how this varies with the parameter α, and find its the value of $\alpha \approx 3.0$ that provides the best model.

In [20]:

```
print('Lasso regression: effect of alpha regularization\n\
parameter on number of features kept in final model\n')

for alpha in [0.5, 1, 2, 3, 5, 10, 20, 50]:
    linlasso = Lasso(alpha, max_iter = 10000).fit(X_train_scaled, y_train)
    r2_train = linlasso.score(X_train_scaled, y_train)
    r2_test = linlasso.score(X_test_scaled, y_test)

    print('Alpha = {:.2f}\nFeatures kept: {}, r-squared training: {:.2f}, \
r-squared test: {:.2f}\n'
         .format(alpha, np.sum(linlasso.coef_ != 0), r2_train, r2_test))
```

```
Lasso regression: effect of alpha regularization
parameter on number of features kept in final model

Alpha = 0.50
Features kept: 35, r-squared training: 0.65, r-squared test: 0.58

Alpha = 1.00
Features kept: 25, r-squared training: 0.64, r-squared test: 0.60

Alpha = 2.00
Features kept: 20, r-squared training: 0.63, r-squared test: 0.62

Alpha = 3.00
Features kept: 17, r-squared training: 0.62, r-squared test: 0.63

Alpha = 5.00
Features kept: 12, r-squared training: 0.60, r-squared test: 0.61

Alpha = 10.00
Features kept: 6, r-squared training: 0.57, r-squared test: 0.58

Alpha = 20.00
Features kept: 2, r-squared training: 0.51, r-squared test: 0.50

Alpha = 50.00
Features kept: 1, r-squared training: 0.31, r-squared test: 0.30
```

Polynomial regression

In [21]:

```
from sklearn.linear_model import LinearRegression
from sklearn.linear_model import Ridge
from sklearn.preprocessing import PolynomialFeatures

###### -------------- LEAST SQUARES REGRESSION---------------------
X_train, X_test, y_train, y_test = train_test_split(X_F1, y_F1,
                                                   random_state = 0)
linreg = LinearRegression().fit(X_train, y_train)

print('linear model coeff (w): {}'
     .format(linreg.coef_))
print('linear model intercept (b): {:.3f}'
     .format(linreg.intercept_))
print('R-squared score (training): {:.3f}'
     .format(linreg.score(X_train, y_train)))
print('R-squared score (test): {:.3f}'
     .format(linreg.score(X_test, y_test)))

###### -------------- POLYNOMIAL LEAST SQUARES REGRESSION---------------------
print('\nNow we transform the original input data to add\n\
polynomial features up to degree 2 (quadratic)\n')
poly = PolynomialFeatures(degree=2) # creates a Polynomialfeatures object on the
original XF1 features
# to produce the new polynomial transform xf1 poly using the polynomial Feature
s's fit_transform method.
X_F1_poly = poly.fit_transform(X_F1)

X_train, X_test, y_train, y_test = train_test_split(X_F1_poly, y_F1,
                                                   random_state = 0)
linreg = LinearRegression().fit(X_train, y_train)

print('(poly deg 2) linear model coeff (w):\n{}'
     .format(linreg.coef_))
print('(poly deg 2) linear model intercept (b): {:.3f}'
     .format(linreg.intercept_))
print('(poly deg 2) R-squared score (training): {:.3f}'
     .format(linreg.score(X_train, y_train)))
print('(poly deg 2) R-squared score (test): {:.3f}\n'
     .format(linreg.score(X_test, y_test)))
###### -------------- POLYNOMIAL LEAST SQUARES REGRESSION WITH REGULARISATION --
-------------------
print('\nAddition of many polynomial features often leads to\n\
overfitting, so we often use polynomial features in combination\n\
with regression that has a regularization penalty, like ridge\n\
regression.\n')

X_train, X_test, y_train, y_test = train_test_split(X_F1_poly, y_F1,
                                                   random_state = 0)
linreg = Ridge().fit(X_train, y_train)

print('(poly deg 2 + ridge) linear model coeff (w):\n{}'
     .format(linreg.coef_))
print('(poly deg 2 + ridge) linear model intercept (b): {:.3f}'
     .format(linreg.intercept_))
print('(poly deg 2 + ridge) R-squared score (training): {:.3f}'
     .format(linreg.score(X_train, y_train)))
print('(poly deg 2 + ridge) R-squared score (test): {:.3f}'
     .format(linreg.score(X_test, y_test)))
```

```
linear model coeff (w): [  4.42   6.     0.53  10.24   6.55  -2.02
-0.32]
linear model intercept (b): 1.543
R-squared score (training): 0.722
R-squared score (test): 0.722

Now we transform the original input data to add
polynomial features up to degree 2 (quadratic)

(poly deg 2) linear model coeff (w):
[  3.41e-12   1.66e+01   2.67e+01  -2.21e+01   1.24e+01   6.93e+00
   1.05e+00   3.71e+00  -1.34e+01  -5.73e+00   1.62e+00   3.66e+00
   5.05e+00  -1.46e+00   1.95e+00  -1.51e+01   4.87e+00  -2.97e+00
  -7.78e+00   5.15e+00  -4.65e+00   1.84e+01  -2.22e+00   2.17e+00
  -1.28e+00   1.88e+00   1.53e-01   5.62e-01  -8.92e-01  -2.18e+00
   1.38e+00  -4.90e+00  -2.24e+00   1.38e+00  -5.52e-01  -1.09e+00]
(poly deg 2) linear model intercept (b): -3.206
(poly deg 2) R-squared score (training): 0.969
(poly deg 2) R-squared score (test): 0.805


Addition of many polynomial features often leads to
overfitting, so we often use polynomial features in combination
with regression that has a regularization penalty, like ridge
regression.

(poly deg 2 + ridge) linear model coeff (w):
[ 0.    2.23  4.73 -3.15  3.86  1.61 -0.77 -0.15 -1.75  1.6   1.37
2.52
  2.72  0.49 -1.94 -1.63  1.51  0.89  0.26  2.05 -1.93  3.62 -0.72
0.63
 -3.16  1.29  3.55  1.73  0.94 -0.51  1.7  -1.98  1.81 -0.22  2.88 -
0.89]
(poly deg 2 + ridge) linear model intercept (b): 5.418
(poly deg 2 + ridge) R-squared score (training): 0.826
(poly deg 2 + ridge) R-squared score (test): 0.825
```

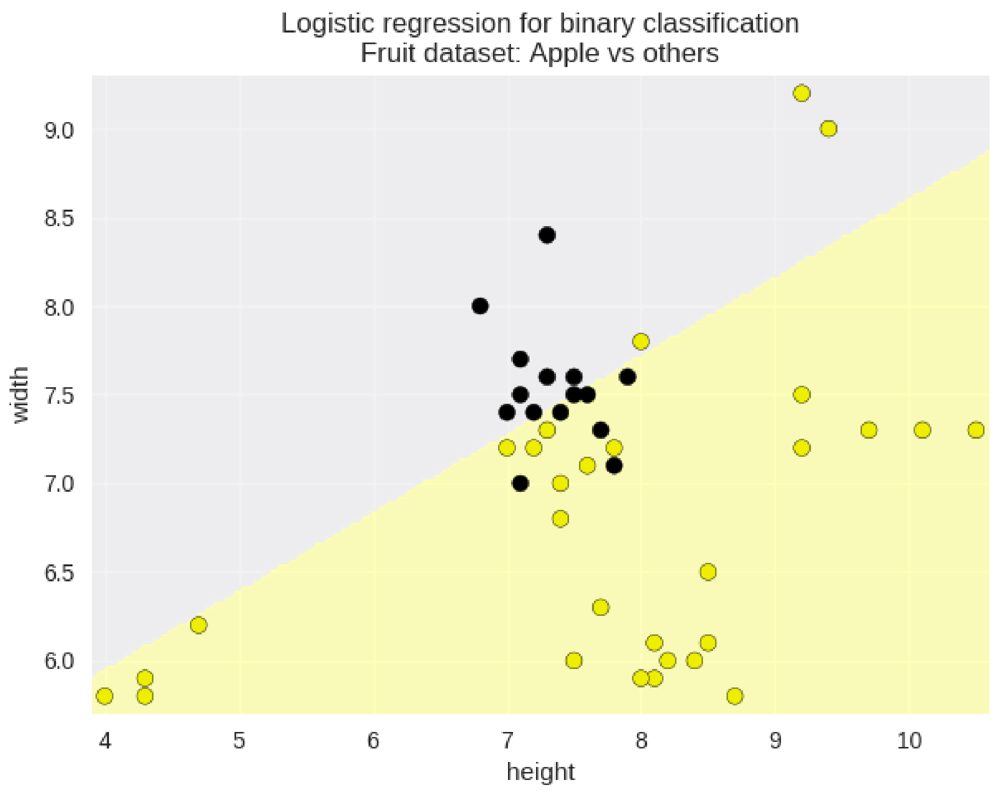
02-07: Linear models for classification: Logistic Regression

Logistic regression for binary classification on fruits dataset using height, width features (positive class: apple, negative class: others)

The dataset used here is a modified form of our fruits dataset, using only height and width as features, and with the target class values modified into a **binary classification problem** predicting whether an object is an apple (positive class) or other fruits (negative class).

In [22]:

```
from sklearn.linear_model import LogisticRegression ## LogisticRegression CLASS
from adspy_shared_utilities import (
plot_class_regions_for_classifier_subplot)

fig, subaxes = plt.subplots(1, 1, figsize=(7, 5))
y_fruits_apple = y_fruits_2d == 1   # make into a binary problem: apples vs ever
ything else
X_train, X_test, y_train, y_test = (
train_test_split(X_fruits_2d.as_matrix(),
                y_fruits_apple.as_matrix(),
                random_state = 0))

clf = LogisticRegression(C=100).fit(X_train, y_train) # create the logistic Regr
ession object by calling the 1
#parameter constructor, and call the fit method on the constructor, assign the t
rained model into the variable clf.
plot_class_regions_for_classifier_subplot(clf, X_train, y_train, None,
                                         None, 'Logistic regression \
for binary classification\nFruit dataset: Apple vs others',
                                         subaxes)

h = 6
w = 8
print('A fruit with height {} and width {} is predicted to be: {}'
     .format(h,w, ['not an apple', 'an apple'][clf.predict([[h,w]])[0]]))
## Note how awesome this convenient this syntax is here - this is why you love p
ython [...] is the list,
## the next square bracket is the indexing operator of the list.

h = 10
w = 7
print('A fruit with height {} and width {} is predicted to be: {}'
     .format(h,w, ['not an apple', 'an apple'][clf.predict([[h,w]])[0]]))
subaxes.set_xlabel('height')
subaxes.set_ylabel('width')

print('Accuracy of Logistic regression classifier on training set: {:.2f}'
     .format(clf.score(X_train, y_train)))
print('Accuracy of Logistic regression classifier on test set: {:.2f}'
     .format(clf.score(X_test, y_test)))
```

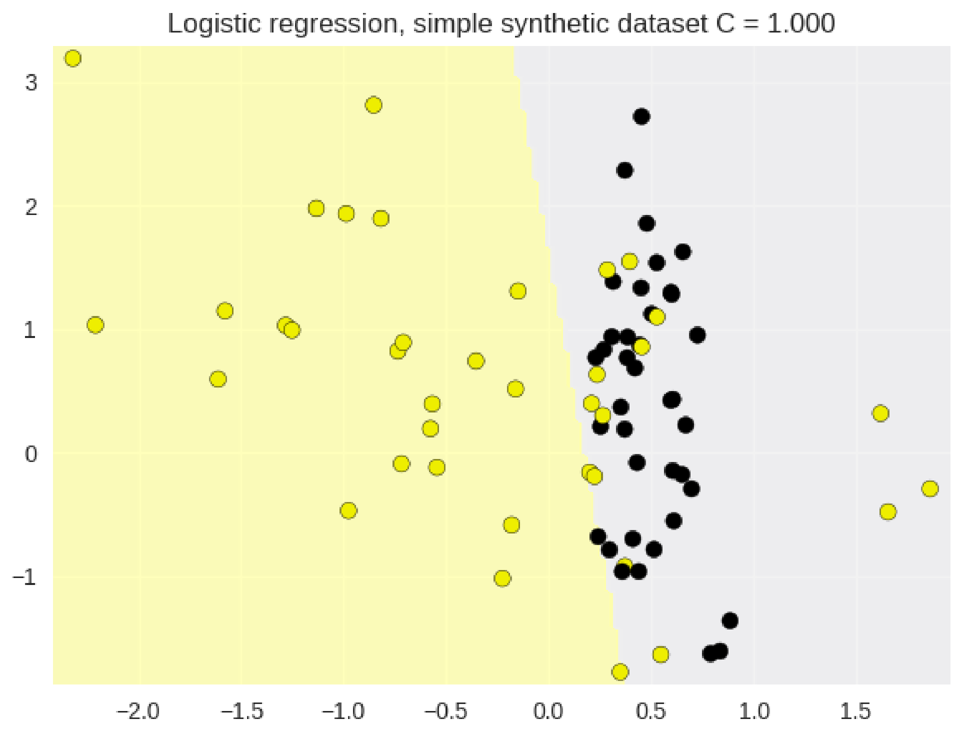


```
A fruit with height 6 and width 8 is predicted to be: an apple
A fruit with height 10 and width 7 is predicted to be: not an apple
Accuracy of Logistic regression classifier on training set: 0.77
Accuracy of Logistic regression classifier on test set: 0.73
```

Logistic regression on simple synthetic dataset

```
In [23]:
```

```
from sklearn.linear_model import LogisticRegression ## LogisticRegression CLASS
from adspy_shared_utilities import (
plot_class_regions_for_classifier_subplot)


X_train, X_test, y_train, y_test = train_test_split(X_C2, y_C2,
                                                    random_state = 0)

fig, subaxes = plt.subplots(1, 1, figsize=(7, 5))
clf = LogisticRegression().fit(X_train, y_train) # call the constructor and call
the fit method
title = 'Logistic regression, simple synthetic dataset C = {:.3f}'.format(1.0)
plot_class_regions_for_classifier_subplot(clf, X_train, y_train,
                                         None, None, title, subaxes)

print('Accuracy of Logistic regression classifier on training set: {:.2f}'
     .format(clf.score(X_train, y_train)))
print('Accuracy of Logistic regression classifier on test set: {:.2f}'
     .format(clf.score(X_test, y_test)))
```



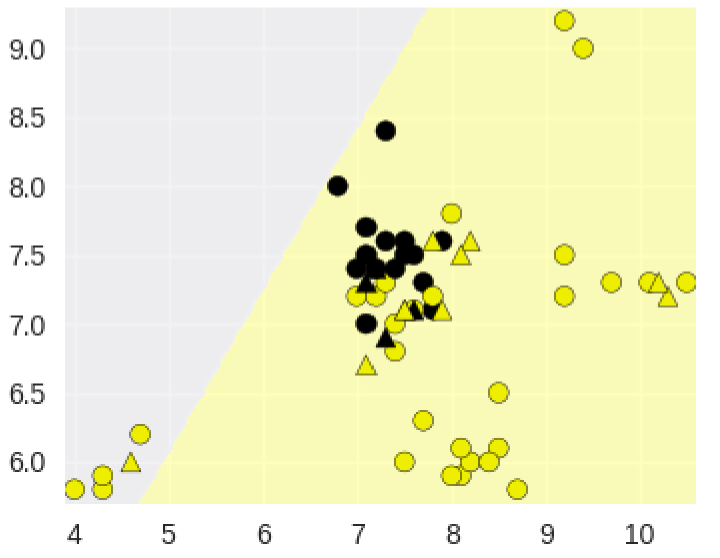
```
Accuracy of Logistic regression classifier on training set: 0.80
Accuracy of Logistic regression classifier on test set: 0.80
```

Logistic regression regularization: C parameter

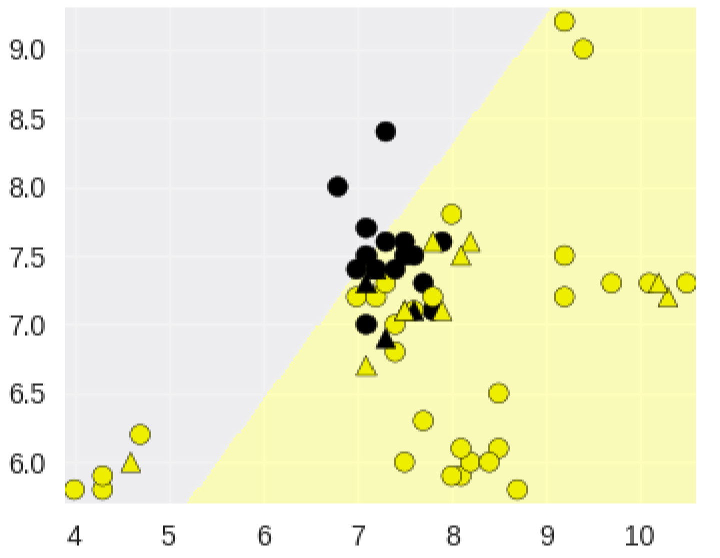
In [24]:

```
X_train, X_test, y_train, y_test = (
train_test_split(X_fruits_2d.as_matrix(),
                y_fruits_apple.as_matrix(),
                random_state=0))

fig, subaxes = plt.subplots(3, 1, figsize=(4, 10))

for this_C, subplot in zip([0.1, 1, 100], subaxes):
    clf = LogisticRegression(C=this_C).fit(X_train, y_train)
    title ='Logistic regression (apple vs rest), C = {:.3f}'.format(this_C)

    plot_class_regions_for_classifier_subplot(clf, X_train, y_train,
                                             X_test, y_test, title,
                                             subplot)
plt.tight_layout()
```

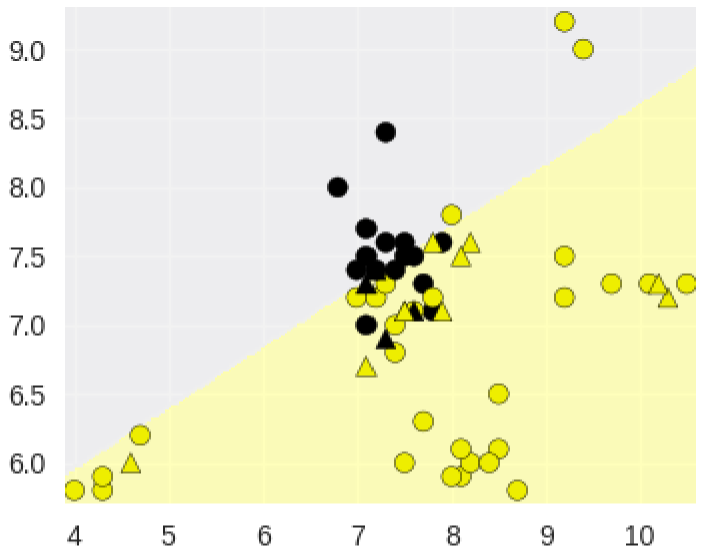
Logistic regression (apple vs rest), C = 0.100
Train score = 0.57, Test score = 0.67



Logistic regression (apple vs rest), C = 1.000
Train score = 0.68, Test score = 0.60



Logistic regression (apple vs rest), C = 100.000
Train score = 0.77, Test score = 0.73



Application to real dataset

```
In [25]:
```

```python
from sklearn.linear_model import LogisticRegression

X_train, X_test, y_train, y_test = train_test_split(X_cancer, y_cancer, random_s
tate = 0)

clf = LogisticRegression().fit(X_train, y_train)
print('Breast cancer dataset')
print('Accuracy of Logistic regression classifier on training set: {:.2f}'
     .format(clf.score(X_train, y_train)))
print('Accuracy of Logistic regression classifier on test set: {:.2f}'
     .format(clf.score(X_test, y_test))) #WOW!
```

```
Breast cancer dataset
Accuracy of Logistic regression classifier on training set: 0.96
Accuracy of Logistic regression classifier on test set: 0.96
```

02-08: Support Vector Machines

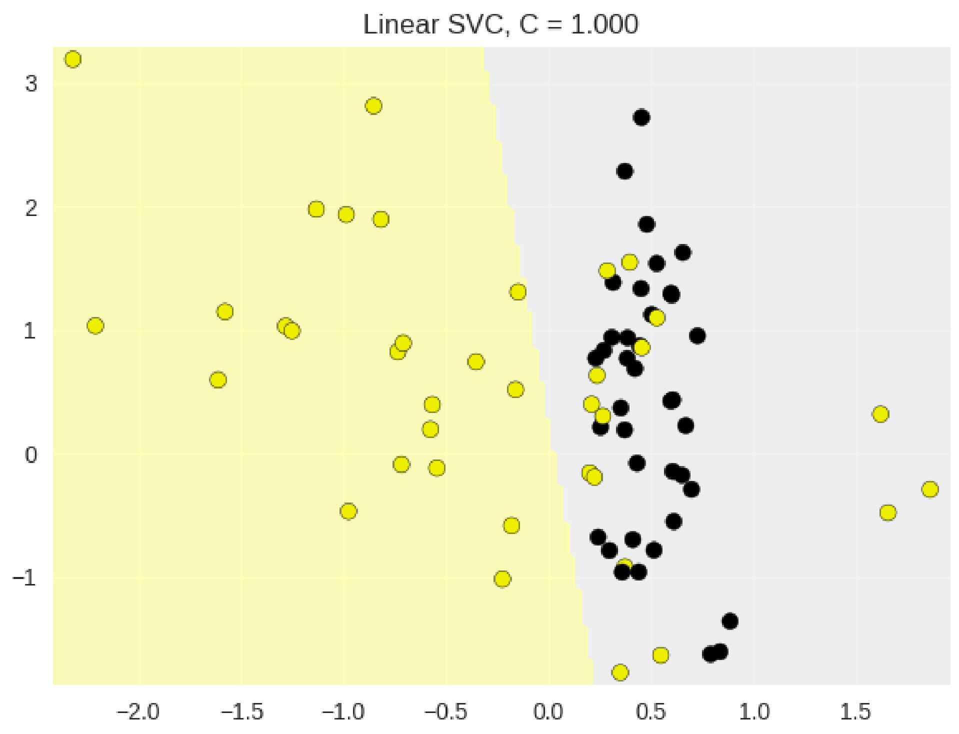
Linear Support Vector Machine

In the example, the two classes were **perfectly separable** with a linear classifier. This however, is not very realistic. In practice though, we typically have noise or just more complexity in the dataset that makes a perfect linear separation impossible, but where most points can be separated without errors by a linear classifier.

How tolerant the SVM is of misclassifying training points as compared to its objective of minimising the margin between classes is controlled by a regularisation parameter called C, which is set to 1.0 by default.

In [26]:

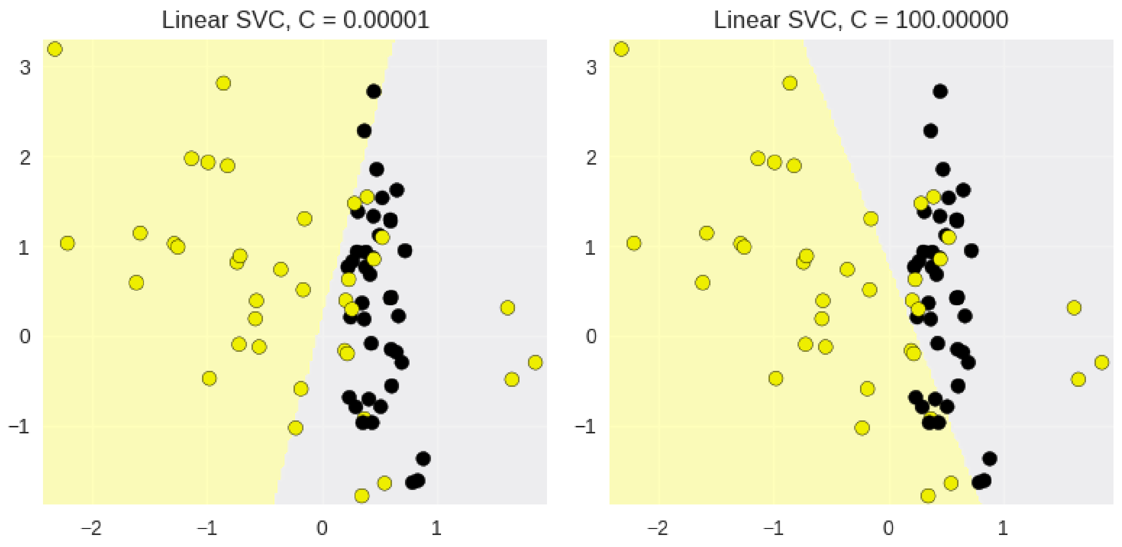
```
from sklearn.svm import SVC #linear SV classifier.
from adspy_shared_utilities import plot_class_regions_for_classifier_subplot


X_train, X_test, y_train, y_test = train_test_split(X_C2, y_C2, random_state = 0
)

fig, subaxes = plt.subplots(1, 1, figsize=(7, 5))
this_C = 1.0
clf = SVC(kernel = 'linear', C=this_C).fit(X_train, y_train)
title = 'Linear SVC, C = {:.3f}'.format(this_C)
plot_class_regions_for_classifier_subplot(clf, X_train, y_train, None, None, tit
le, subaxes)
```



Linear Support Vector Machine: Varying the C parameter

In [27]:

```
from sklearn.svm import LinearSVC
from adspy_shared_utilities import plot_class_regions_for_classifier

X_train, X_test, y_train, y_test = train_test_split(X_C2, y_C2, random_state = 0
)
fig, subaxes = plt.subplots(1, 2, figsize=(8, 4))

for this_C, subplot in zip([0.00001, 100], subaxes):
    clf = LinearSVC(C=this_C).fit(X_train, y_train)
    title = 'Linear SVC, C = {:.5f}'.format(this_C)
    plot_class_regions_for_classifier_subplot(clf, X_train, y_train,
                                              None, None, title, subplot)
plt.tight_layout()
```



Application to real dataset

In [28]:

```
from sklearn.svm import LinearSVC
X_train, X_test, y_train, y_test = train_test_split(X_cancer, y_cancer, random_s
tate = 0)

clf = LinearSVC().fit(X_train, y_train)
print('Breast cancer dataset')
print('Accuracy of Linear SVC classifier on training set: {:.2f}'
     .format(clf.score(X_train, y_train)))
print('Accuracy of Linear SVC classifier on test set: {:.2f}'
     .format(clf.score(X_test, y_test)))
```

```
Breast cancer dataset
Accuracy of Linear SVC classifier on training set: 0.93
Accuracy of Linear SVC classifier on test set: 0.94
```

02-09: Multi-class classification with linear models

LinearSVC with M classes generates M one vs rest classifiers.

In [29]:

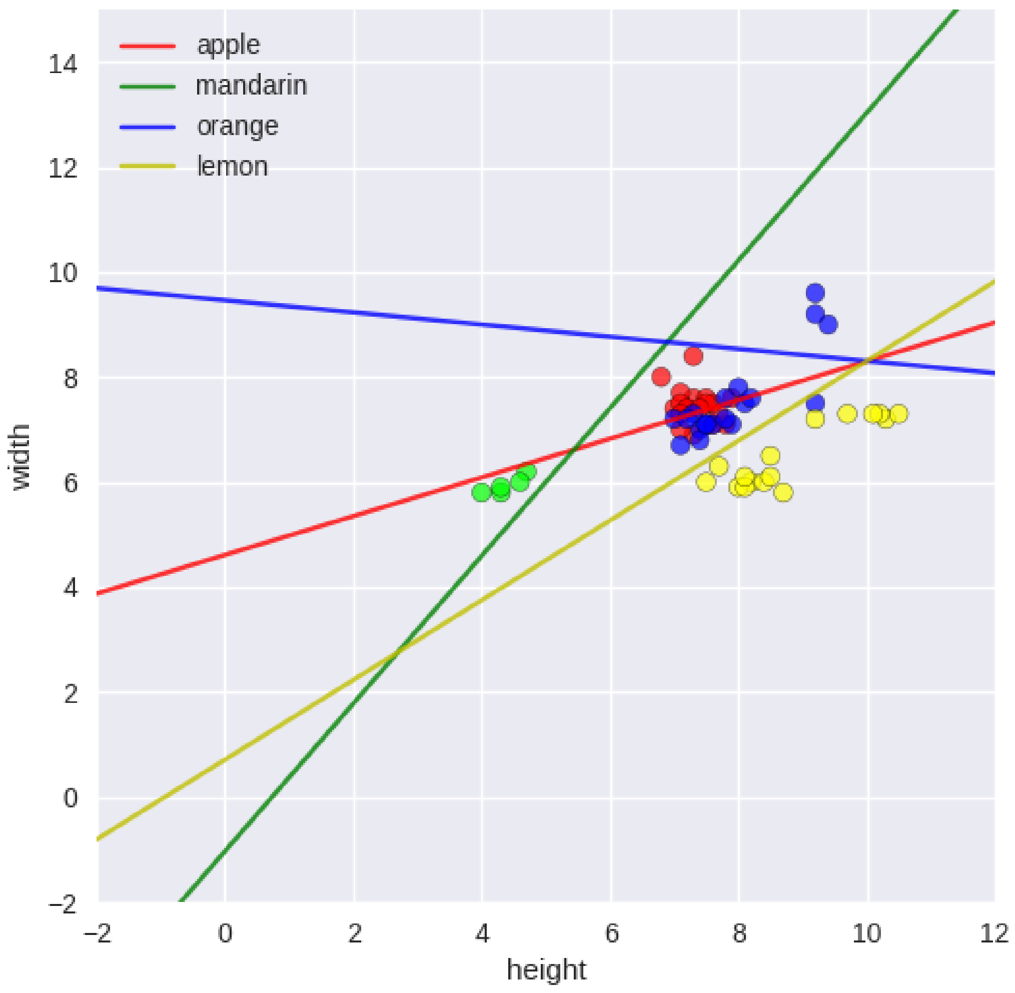
```
from sklearn.svm import LinearSVC

X_train, X_test, y_train, y_test = train_test_split(X_fruits_2d, y_fruits_2d, ra
ndom_state = 0)

clf = LinearSVC(C=5, random_state = 67).fit(X_train, y_train)
print('Coefficients:\n', clf.coef_)
print('Intercepts:\n', clf.intercept_)
```

```
Coefficients:
 [[-0.26  0.71]
 [-1.63  1.16]
 [ 0.03  0.29]
 [ 1.24 -1.64]]
Intercepts:
 [-3.29  1.2  -2.72  1.16]
```

Multi-class results on the fruit dataset

In [30]:

```
plt.figure(figsize=(6,6))
colors = ['r', 'g', 'b', 'y']
cmap_fruits = ListedColormap(['#FF0000', '#00FF00', '#0000FF','#FFFF00'])

plt.scatter(X_fruits_2d[['height']], X_fruits_2d[['width']],
           c=y_fruits_2d, cmap=cmap_fruits, edgecolor = 'black', alpha=.7)

x_0_range = np.linspace(-10, 15)

for w, b, color in zip(clf.coef_, clf.intercept_, ['r', 'g', 'b', 'y']):
    # Since class prediction with a linear model uses the formula y = w_0 x_0 +
 w_1 x_1 + b,
    # and the decision boundary is defined as being all points with y = 0, to pl
ot x_1 as a
    # function of x_0 we just solve w_0 x_0 + w_1 x_1 + b = 0 for x_1:
    plt.plot(x_0_range, -(x_0_range * w[0] + b) / w[1], c=color, alpha=.8)

plt.legend(target_names_fruits)
plt.xlabel('height')
plt.ylabel('width')
plt.xlim(-2, 12)
plt.ylim(-2, 15)
plt.show()
```

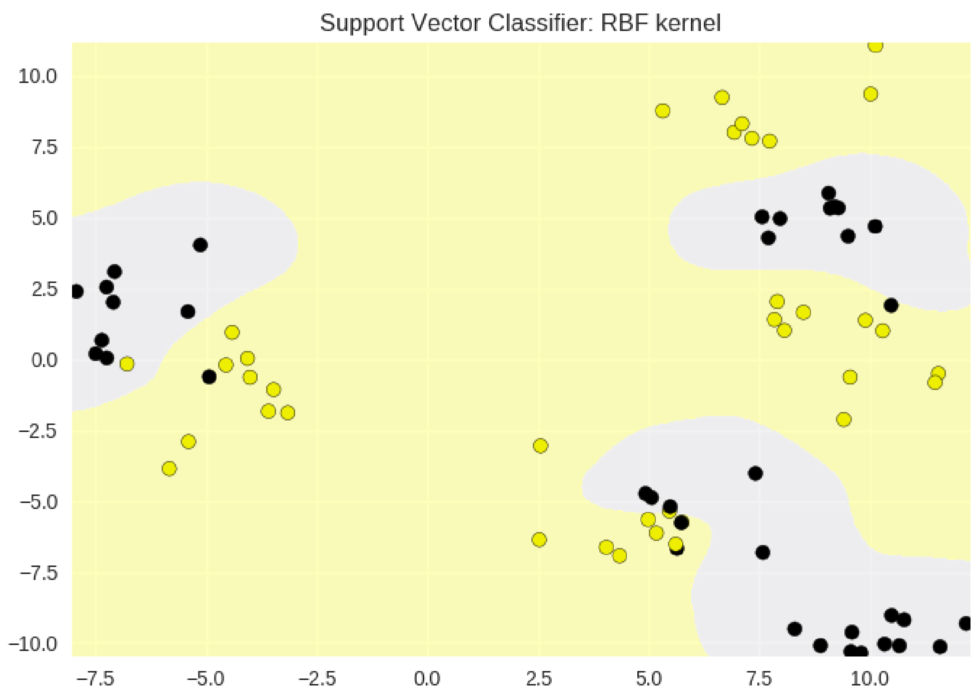


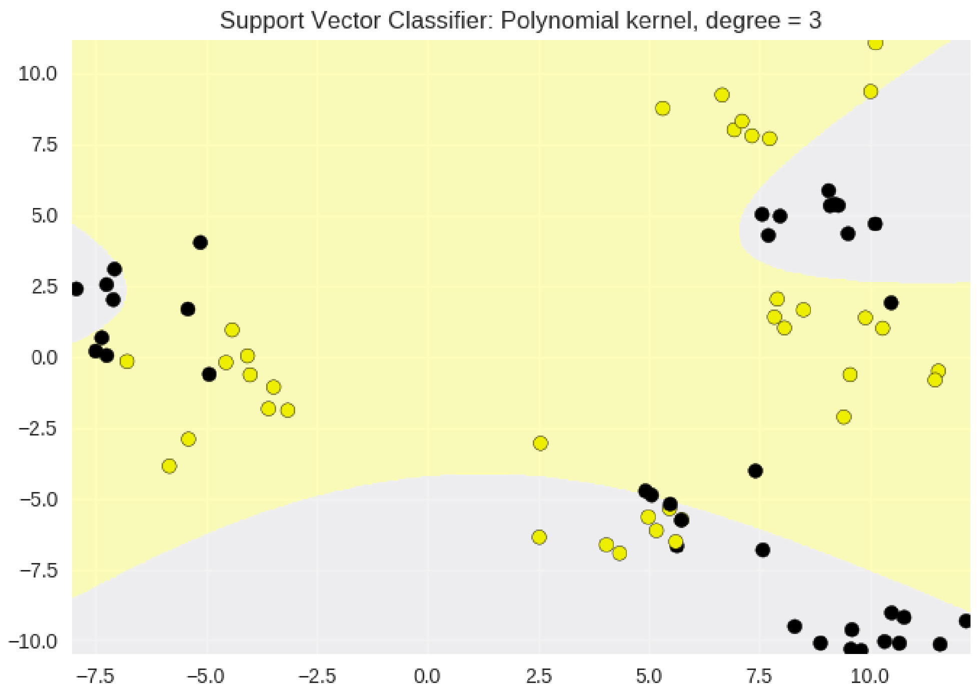
02-10: Kernelized Support Vector Machines

Classification

In [31]:

```
from sklearn.svm import SVC ## Support Vector Class
from adspy_shared_utilities import plot_class_regions_for_classifier

X_train, X_test, y_train, y_test = train_test_split(X_D2, y_D2, random_state = 0
)

# The default SVC kernel is radial basis function (RBF)
plot_class_regions_for_classifier(SVC().fit(X_train, y_train),
                                 X_train, y_train, None, None,
                                 'Support Vector Classifier: RBF kernel')

# Compare decision boundries with polynomial kernel, degree = 3
plot_class_regions_for_classifier(SVC(kernel = 'poly', degree = 3) #it takes in
 an argument called "kernel" and a
                                  # additional parameter degree
                                  .fit(X_train, y_train), X_train, # by default
 it will use the radial basis function,
                                 y_train, None, None, # but a number of other ch
oices are supported!
                                 'Support Vector Classifier: Polynomial kernel,
 degree = 3')
```


Support Vector Classifier: RBF kernel
10.0
7.5
5.0
2.5
0.0
−2.5
−5.0
−7.5
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0

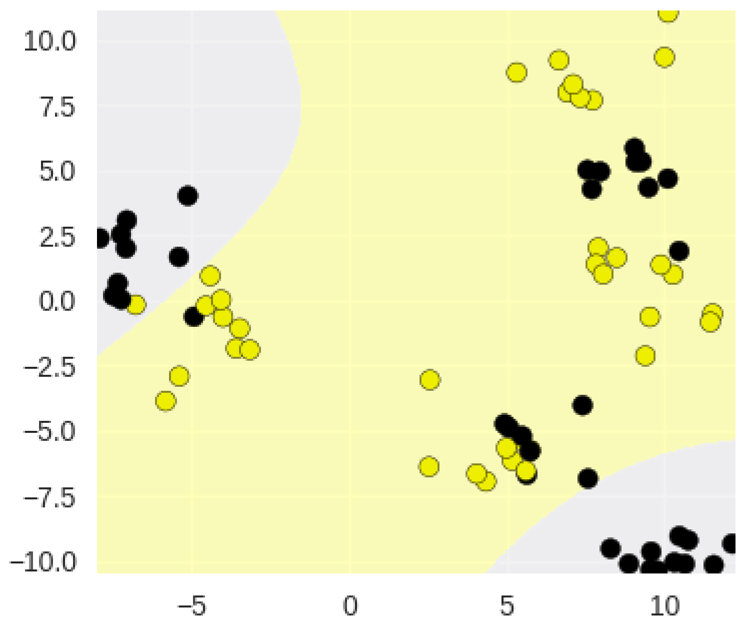


Support Vector Machine with RBF kernel: gamma parameter

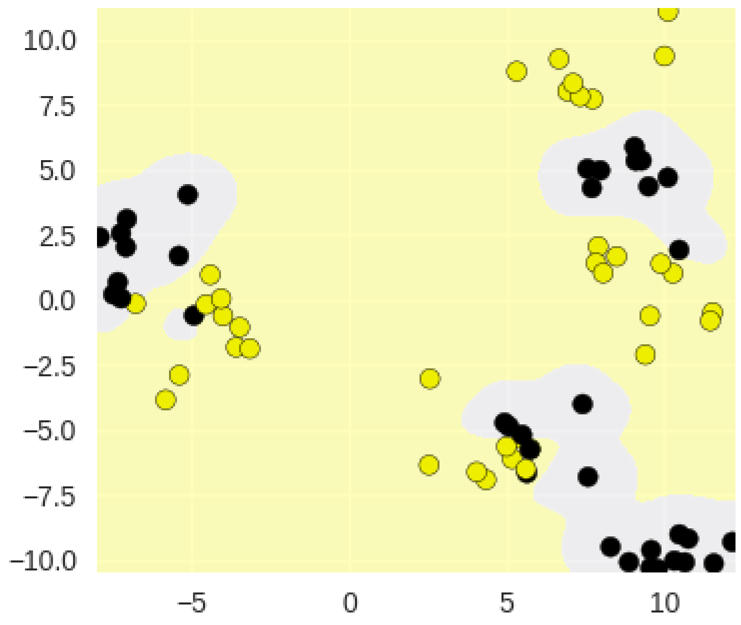
In [32]:

```
from adspy_shared_utilities import plot_class_regions_for_classifier

X_train, X_test, y_train, y_test = train_test_split(X_D2, y_D2, random_state = 0
)
fig, subaxes = plt.subplots(3, 1, figsize=(4, 11))

for this_gamma, subplot in zip([0.01, 1.0, 10.0], subaxes):
    clf = SVC(kernel = 'rbf', gamma=this_gamma).fit(X_train, y_train)
    title = 'Support Vector Classifier: \nRBF kernel, gamma = {:.2f}'.format(thi
s_gamma)
    plot_class_regions_for_classifier_subplot(clf, X_train, y_train,
                                             None, None, title, subplot)
    plt.tight_layout()
```

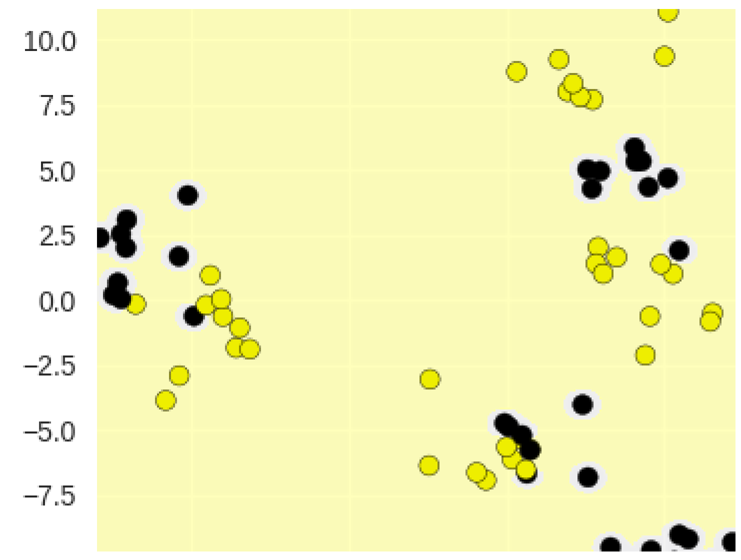
Support Vector Classifier:
RBF kernel, gamma = 0.01

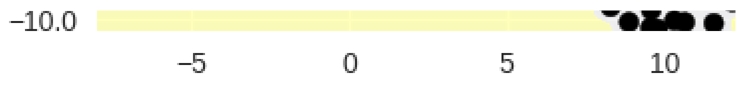


Support Vector Classifier:
RBF kernel, gamma = 1.00



Support Vector Classifier:
RBF kernel, gamma = 10.00

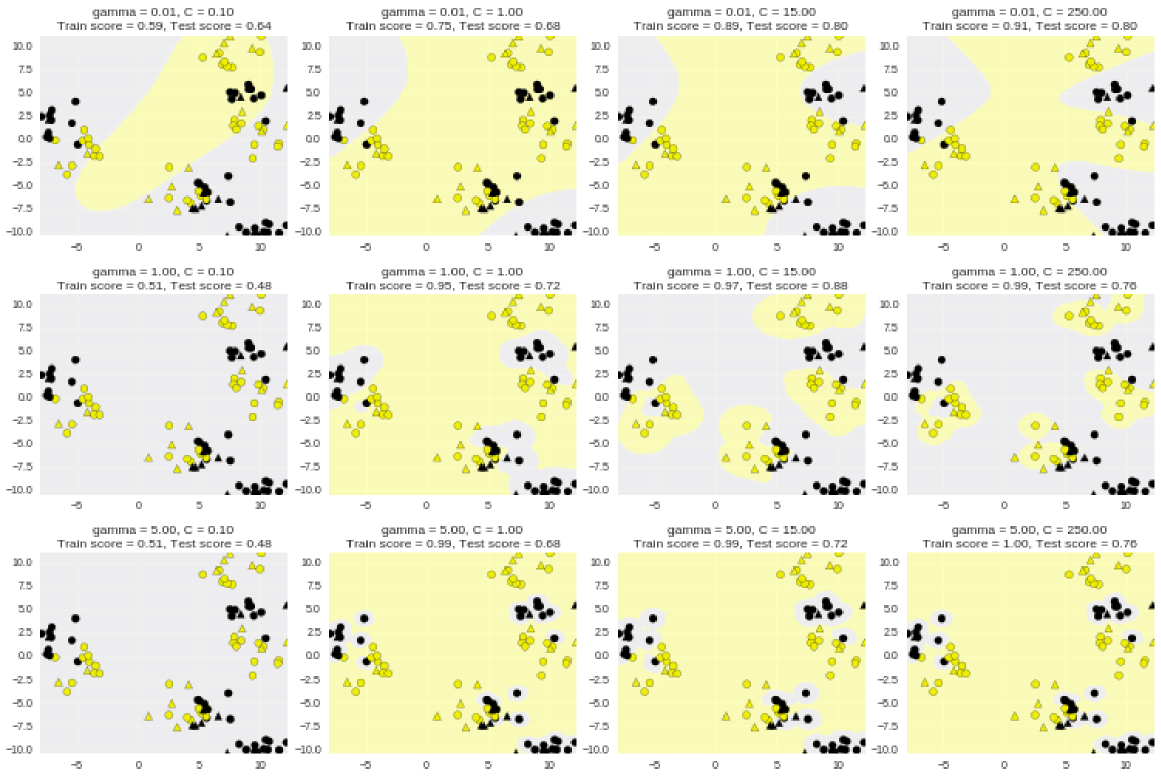




Support Vector Machine with RBF kernel: using both C and gamma parameter

In [33]:

```
from sklearn.svm import SVC
from adspy_shared_utilities import plot_class_regions_for_classifier_subplot

from sklearn.model_selection import train_test_split


X_train, X_test, y_train, y_test = train_test_split(X_D2, y_D2, random_state = 0
)
fig, subaxes = plt.subplots(3, 4, figsize=(15, 10), dpi=50)

for this_gamma, this_axis in zip([0.01, 1, 5], subaxes):

    for this_C, subplot in zip([0.1, 1, 15, 250], this_axis):
        title = 'gamma = {:.2f}, C = {:.2f}'.format(this_gamma, this_C)
        clf = SVC(kernel = 'rbf', gamma = this_gamma,
                  C = this_C).fit(X_train, y_train)
        plot_class_regions_for_classifier_subplot(clf, X_train, y_train,
                                                  X_test, y_test, title,
                                                  subplot)
        plt.tight_layout(pad=0.4, w_pad=0.5, h_pad=1.0)
```



Application of SVMs to a real dataset: unnormalized data

In [34]:

```
from sklearn.svm import SVC
X_train, X_test, y_train, y_test = train_test_split(X_cancer, y_cancer,
                                                    random_state = 0)

clf = SVC(C=10).fit(X_train, y_train)
print('Breast cancer dataset (unnormalized features)')
print('Accuracy of RBF-kernel SVC on training set: {:.2f}'
     .format(clf.score(X_train, y_train)))
print('Accuracy of RBF-kernel SVC on test set: {:.2f}'
     .format(clf.score(X_test, y_test)))
```

```
Breast cancer dataset (unnormalized features)
Accuracy of RBF-kernel SVC on training set: 1.00
Accuracy of RBF-kernel SVC on test set: 0.63
```

The SVM is overfitting - by performing too well on the training data and much poorer on the test data.

Application of SVMs to a real dataset: normalized data with feature preprocessing using minmax scaling

In [35]:

```
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

clf = SVC(C=10).fit(X_train_scaled, y_train)
print('Breast cancer dataset (normalized with MinMax scaling)')
print('RBF-kernel SVC (with MinMax scaling) training set accuracy: {:.2f}'
     .format(clf.score(X_train_scaled, y_train)))
print('RBF-kernel SVC (with MinMax scaling) test set accuracy: {:.2f}'
     .format(clf.score(X_test_scaled, y_test)))
```

```
Breast cancer dataset (normalized with MinMax scaling)
RBF-kernel SVC (with MinMax scaling) training set accuracy: 0.98
RBF-kernel SVC (with MinMax scaling) test set accuracy: 0.96
```

02-11: Cross-validation

Example based on k-NN classifier with fruit dataset (2 features)

In [37]:

```
from sklearn.model_selection import cross_val_score ##for cross validation

clf = KNeighborsClassifier(n_neighbors = 5)
X = X_fruits_2d.as_matrix()
y = y_fruits_2d.as_matrix()
cv_scores = cross_val_score(clf, X, y)
#parameter order: model you want to evaluate (clf), dataset (X), corresponding t
ruth target labels or values.
#by default this does a 3-fold cross validation, so it gives 3 accuracy scores
# if you would like to change the number of folds, include the cv parameter, whe
re cv = 10 will perform a 10 fold cross
# validation.

print('Cross-validation scores (3-fold):', cv_scores) # 3 accuracy scores in a l
ist.
print('Mean cross-validation score (3-fold): {:.3f}'
     .format(np.mean(cv_scores))) # and you can calculate the mean here.
```

```
Cross-validation scores (3-fold): [ 0.77  0.74  0.83]
Mean cross-validation score (3-fold): 0.781
```

This gives us an indication of how sensitive the model is to the nature of the specific training set. We can look at the distribution of these multiple scores across all the cross-validation folds to see how likely it is that by chance, the model will perform very well or very poorly on any given dataset - so we can do a worst case or best case performance estimate from these multiple scores.

This extra information does come with extra cost - it takes more time and computation to do cross-validation. If we perform K-fold cross validation and we don't compute the fold results in parallel, it'll take about k times as long to get the accuracy scores as it would with just one Train/Test split.

However, the additional knowledge on how well your model will perform on future data is often **well worth this cost**.

A note on performing cross-validation for more advanced scenarios.

In some cases (e.g. when feature values have very different ranges), we've seen the need to scale or normalize the training and test sets before use with a classifier. The proper way to do cross-validation when you need to scale the data is *not* to scale the entire dataset with a single transform, since this will indirectly leak information into the training data about the whole dataset, including the test data (see the lecture on data leakage later in the course). Instead, scaling/normalizing must be computed and applied for each cross-validation fold separately. To do this, the easiest way in scikit-learn is to use *pipelines*. While these are beyond the scope of this course, further information is available in the scikit-learn documentation here:

http://scikit-learn.org/stable/modules/generated/sklearn.pipeline.Pipeline.html (http://scikit-learn.org/stable/modules/generated/sklearn.pipeline.Pipeline.html)

or the Pipeline section in the recommended textbook: Introduction to Machine Learning with Python by Andreas C. Müller and Sarah Guido (O'Reilly Media).

Validation curve example

The code shown below will fit three models using a RBF SVM on different subsets of the data, corresponding to the three different specified values of the kernel's gamma parameter.

This returns two 4x3 arrays - which is 4 levels of gamma and 3 fits per level containing scores for the training and test sets.

In [38]:

```
from sklearn.svm import SVC
from sklearn.model_selection import validation_curve

param_range = np.logspace(-3, 3, 4)
train_scores, test_scores = validation_curve(SVC(), X, y,
                                             param_name='gamma',
                                             param_range=param_range, cv=3)
```

In [39]:

```
print(train_scores)
```

```
[[ 0.49  0.42  0.41]
 [ 0.84  0.72  0.76]
 [ 0.92  0.9   0.93]
 [ 1.    1.    0.98]]
```

In [40]:

```
print(test_scores)
```

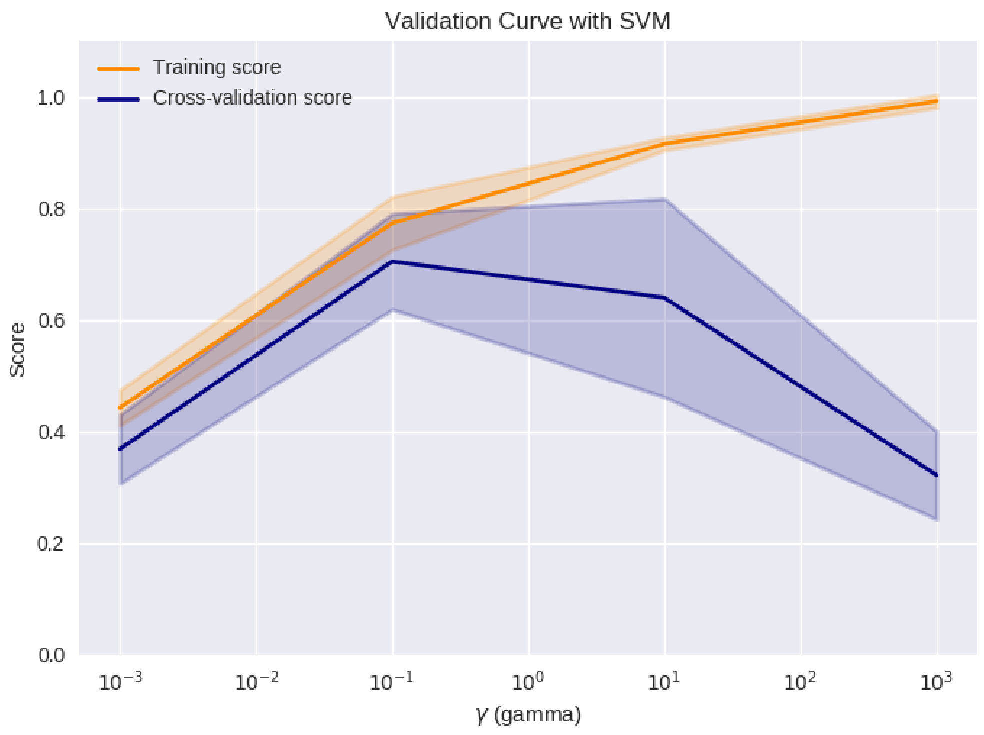
```
[[ 0.45  0.32  0.33]
 [ 0.82  0.68  0.61]
 [ 0.41  0.84  0.67]
 [ 0.36  0.21  0.39]]
```

You can plot these results from the validation curve as shown here to get an idea of how sensitive the performance of the model is to changes in the given parameter. The x axis corresponds to values of the parameter, and the y-axis gives the evaluation score - like the accuracy of the classifier.

In [41]:

```
# This code based on scikit-learn validation_plot example
#  See:  http://scikit-learn.org/stable/auto_examples/model_selection/plot_valid
ation_curve.html
plt.figure()

train_scores_mean = np.mean(train_scores, axis=1)
train_scores_std = np.std(train_scores, axis=1)
test_scores_mean = np.mean(test_scores, axis=1)
test_scores_std = np.std(test_scores, axis=1)

plt.title('Validation Curve with SVM')
plt.xlabel('$\gamma$ (gamma)')
plt.ylabel('Score')
plt.ylim(0.0, 1.1)
lw = 2

plt.semilogx(param_range, train_scores_mean, label='Training score',
            color='darkorange', lw=lw)

plt.fill_between(param_range, train_scores_mean - train_scores_std,
                train_scores_mean + train_scores_std, alpha=0.2,
                color='darkorange', lw=lw)

plt.semilogx(param_range, test_scores_mean, label='Cross-validation score',
            color='navy', lw=lw)

plt.fill_between(param_range, test_scores_mean - test_scores_std,
                test_scores_mean + test_scores_std, alpha=0.2,
                color='navy', lw=lw)

plt.legend(loc='best')
plt.show()
```

```
/opt/conda/lib/python3.6/site-packages/matplotlib/pyplot.py:524: Run
timeWarning: More than 20 figures have been opened. Figures created
through the pyplot interface (`matplotlib.pyplot.figure`) are retain
ed until explicitly closed and may consume too much memory. (To cont
rol this warning, see the rcParam `figure.max_open_warning`).
  max_open_warning, RuntimeWarning)
```



Note that cross validation is used to evaluate the model and not learn or tune a new model. To do model tuning, we'll look at how to tune the model's parameters using Grid Search in a later lecture.

02-12: Decision Trees

In [42]:

```
from sklearn.datasets import load_iris
from sklearn.tree import DecisionTreeClassifier
from adspy_shared_utilities import plot_decision_tree
from sklearn.model_selection import train_test_split


iris = load_iris()

X_train, X_test, y_train, y_test = train_test_split(iris.data, iris.target, rand
om_state = 3)
clf = DecisionTreeClassifier().fit(X_train, y_train)

print('Accuracy of Decision Tree classifier on training set: {:.2f}'
     .format(clf.score(X_train, y_train)))
print('Accuracy of Decision Tree classifier on test set: {:.2f}'
     .format(clf.score(X_test, y_test)))
```

```
Accuracy of Decision Tree classifier on training set: 1.00
Accuracy of Decision Tree classifier on test set: 0.95
```

Setting max decision tree depth to help avoid overfitting

In [43]:

```
clf2 = DecisionTreeClassifier(max_depth = 3).fit(X_train, y_train)

print('Accuracy of Decision Tree classifier on training set: {:.2f}'
      .format(clf2.score(X_train, y_train)))
print('Accuracy of Decision Tree classifier on test set: {:.2f}'
      .format(clf2.score(X_test, y_test)))
```

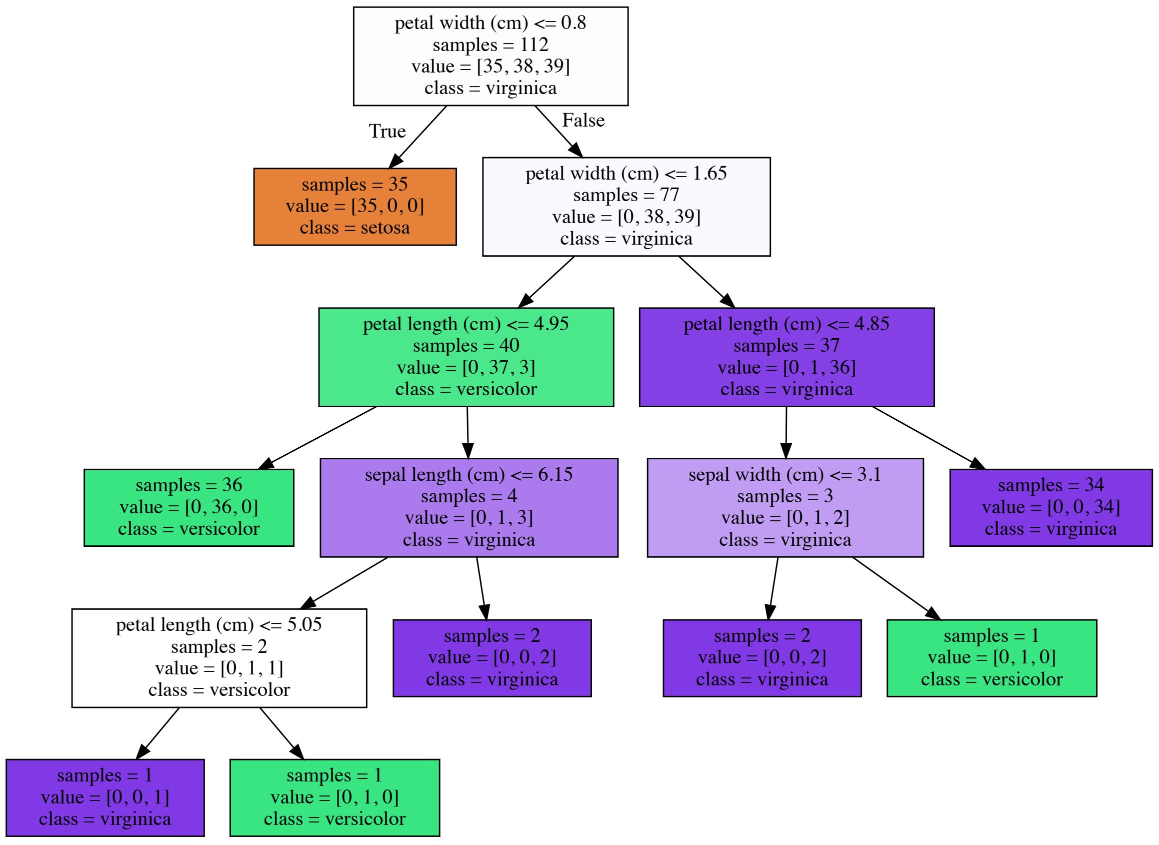
```
Accuracy of Decision Tree classifier on training set: 0.98
Accuracy of Decision Tree classifier on test set: 0.97
```

Visualizing decision trees

In [44]:

```
plot_decision_tree(clf, iris.feature_names, iris.target_names)
```

Out[44]:

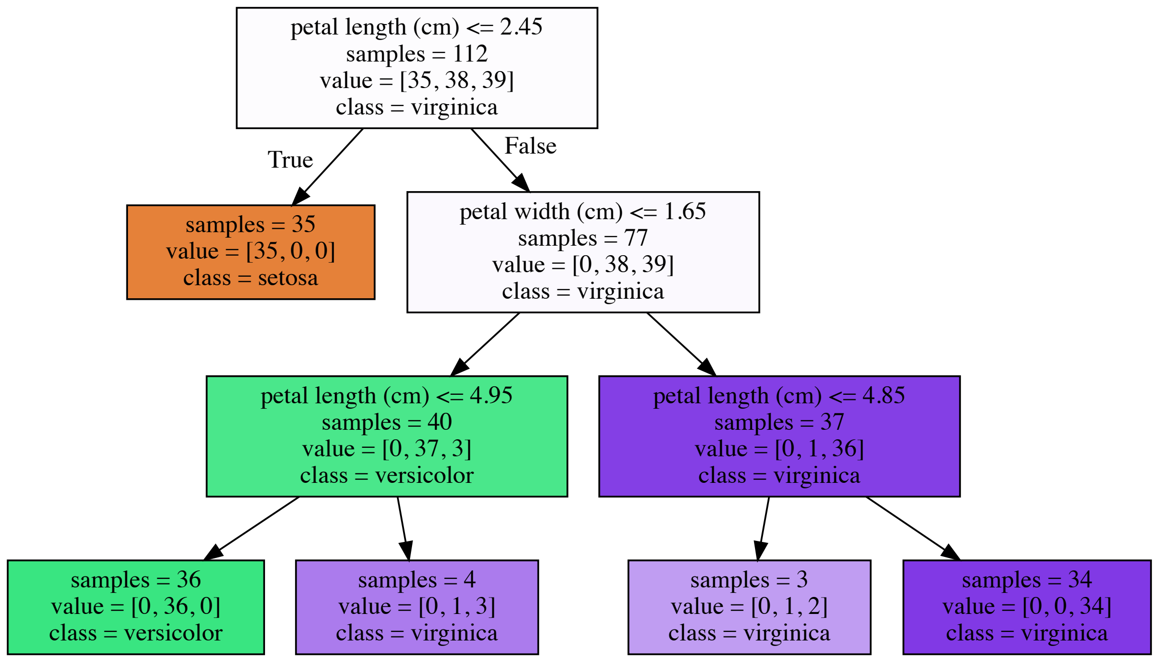


Pre-pruned version (max_depth = 3)

```
In [45]:
```

```
plot_decision_tree(clf2, iris.feature_names, iris.target_names)
```

```
Out[45]:
```

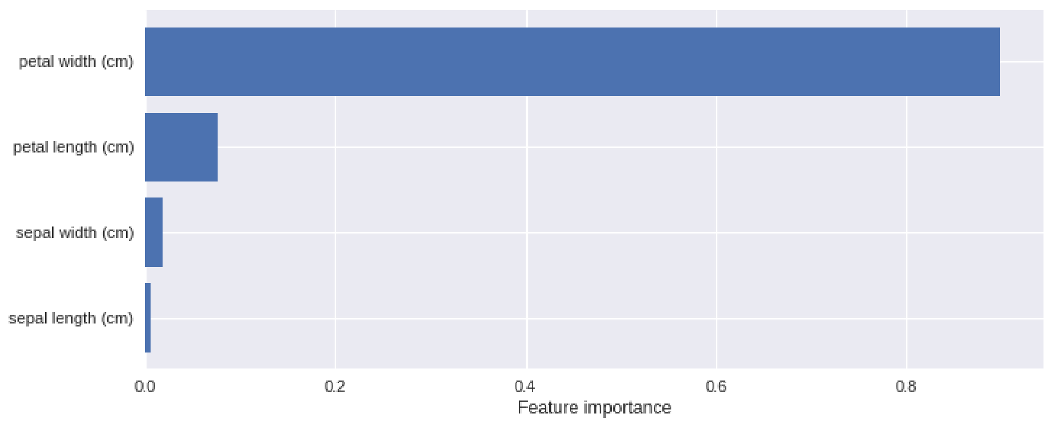


Feature importance

In [46]:

```
from adspy_shared_utilities import plot_feature_importances

plt.figure(figsize=(10,4), dpi=80)
plot_feature_importances(clf, iris.feature_names)
plt.show()

print('Feature importances: {}'.format(clf.feature_importances_))
```

```
/opt/conda/lib/python3.6/site-packages/matplotlib/pyplot.py:524: Run
timeWarning: More than 20 figures have been opened. Figures created
through the pyplot interface (`matplotlib.pyplot.figure`) are retain
ed until explicitly closed and may consume too much memory. (To cont
rol this warning, see the rcParam `figure.max_open_warning`).
  max_open_warning, RuntimeWarning)
```

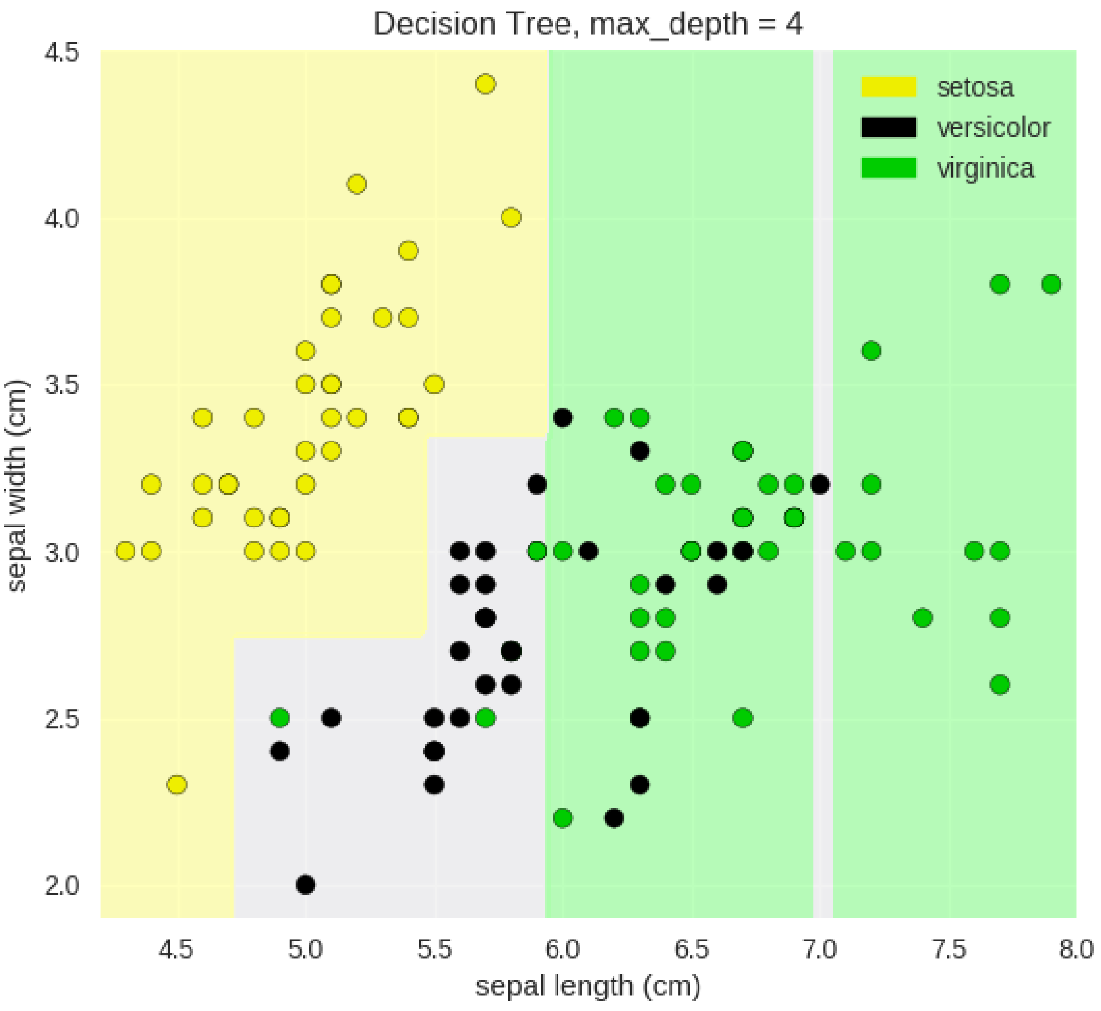


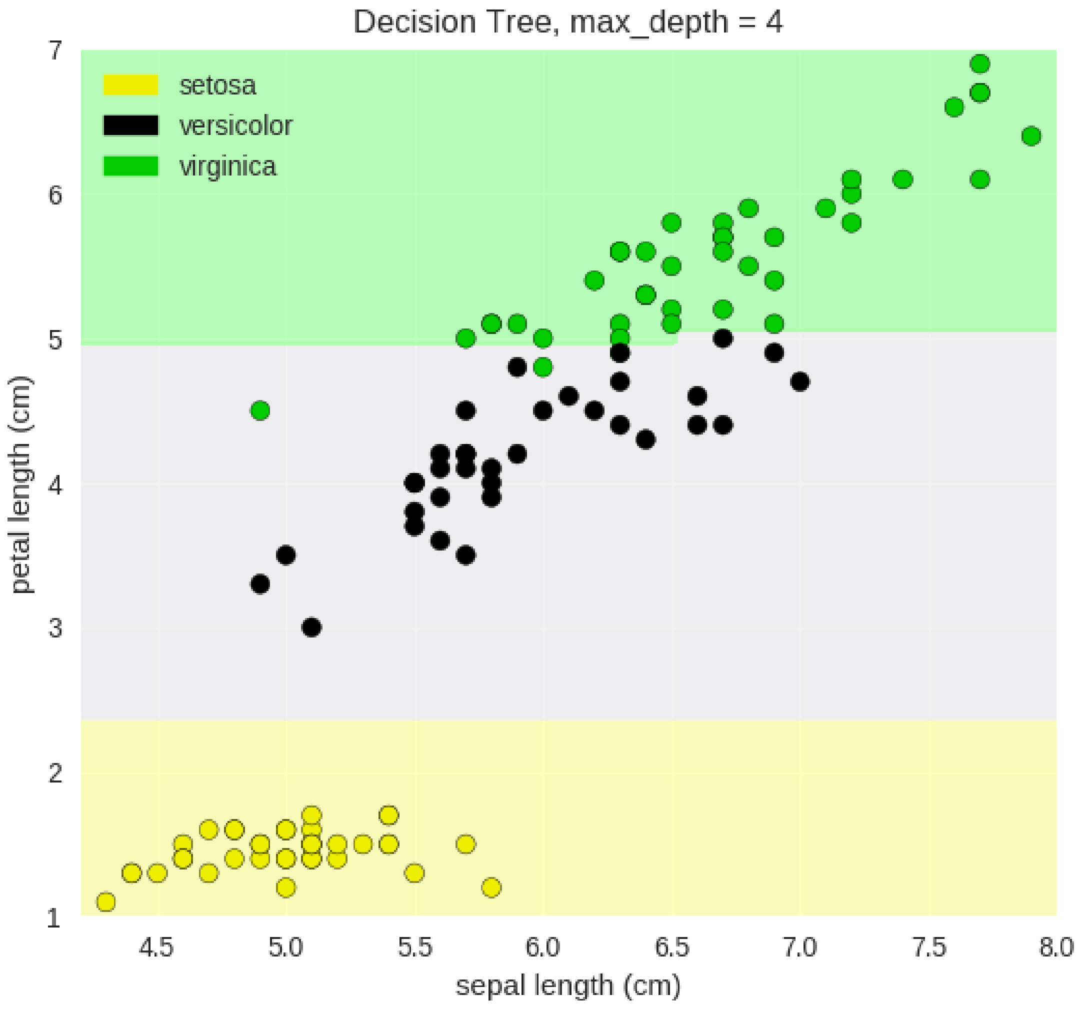
```
Feature importances: [ 0.01  0.02  0.08  0.9 ]
```

In [47]:

```
from sklearn.tree import DecisionTreeClassifier
from adspy_shared_utilities import plot_class_regions_for_classifier_subplot

X_train, X_test, y_train, y_test = train_test_split(iris.data, iris.target, rand
om_state = 0)
fig, subaxes = plt.subplots(6, 1, figsize=(6, 32))

pair_list = [[0,1], [0,2], [0,3], [1,2], [1,3], [2,3]]
tree_max_depth = 4

for pair, axis in zip(pair_list, subaxes):
    X = X_train[:, pair]
    y = y_train

    clf = DecisionTreeClassifier(max_depth=tree_max_depth).fit(X, y)
    title = 'Decision Tree, max_depth = {:d}'.format(tree_max_depth)
    plot_class_regions_for_classifier_subplot(clf, X, y, None,
                                              None, title, axis,
                                              iris.target_names)

    axis.set_xlabel(iris.feature_names[pair[0]])
    axis.set_ylabel(iris.feature_names[pair[1]])

plt.tight_layout()
plt.show()
```

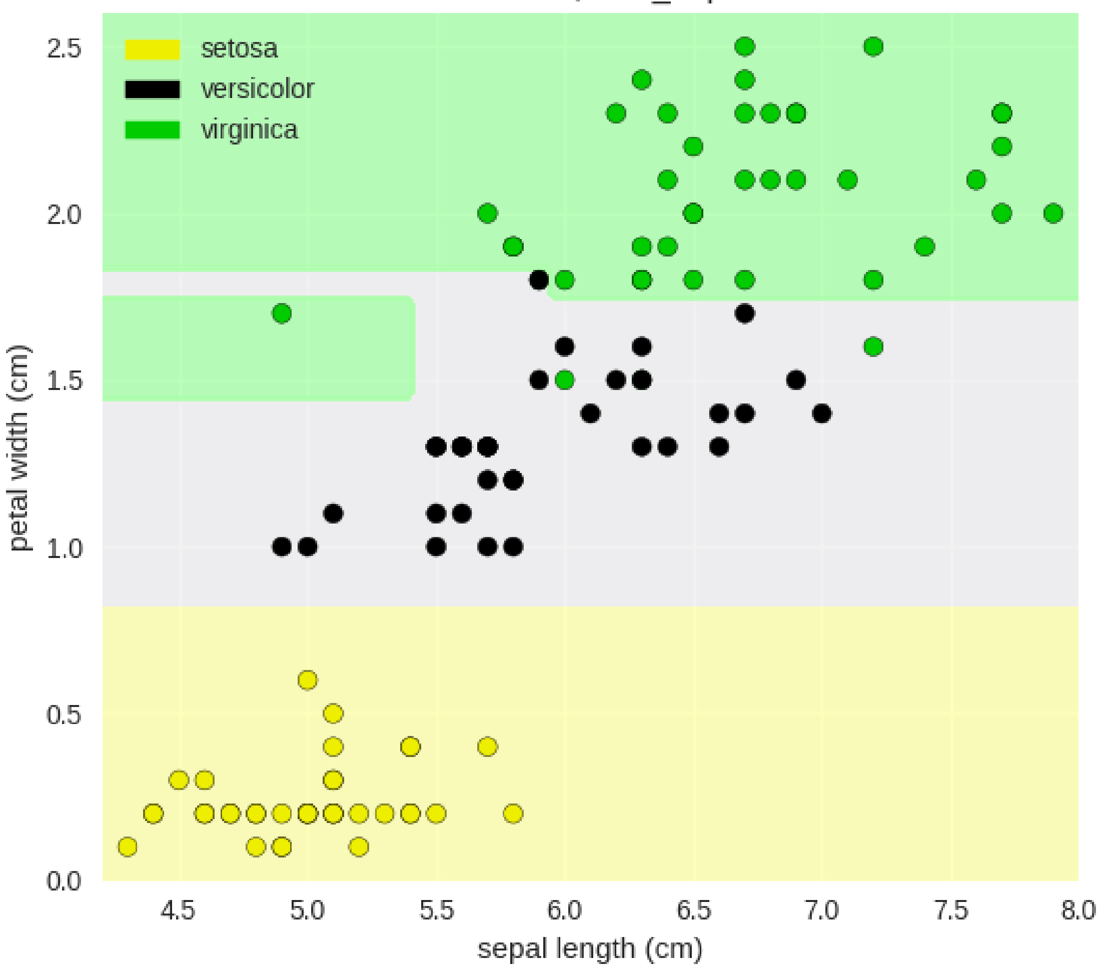
```
/opt/conda/lib/python3.6/site-packages/matplotlib/pyplot.py:524: Run
timeWarning: More than 20 figures have been opened. Figures created
through the pyplot interface (`matplotlib.pyplot.figure`) are retain
ed until explicitly closed and may consume too much memory. (To cont
rol this warning, see the rcParam `figure.max_open_warning`).
  max_open_warning, RuntimeWarning)
```



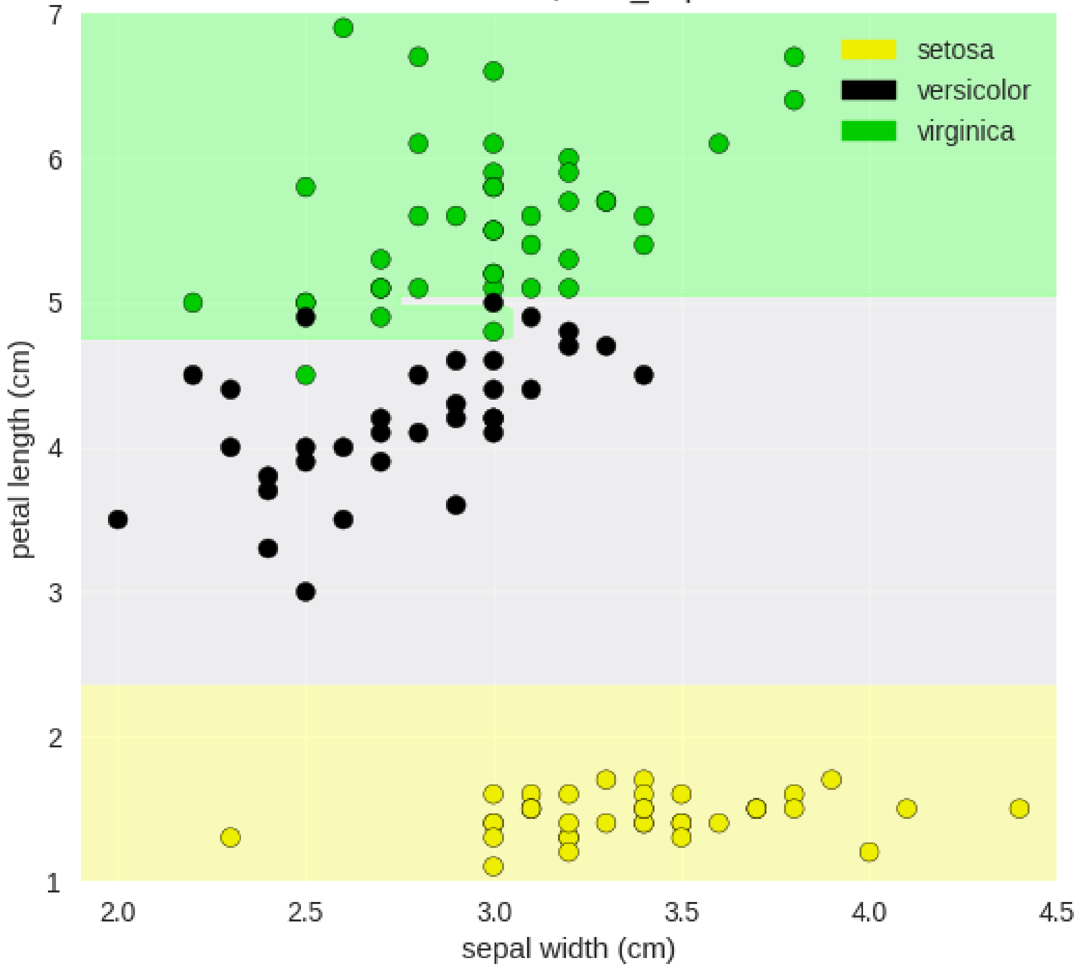


Decision Tree, max_depth = 4

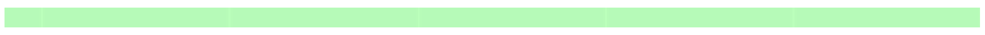
Decision Tree, max_depth = 4

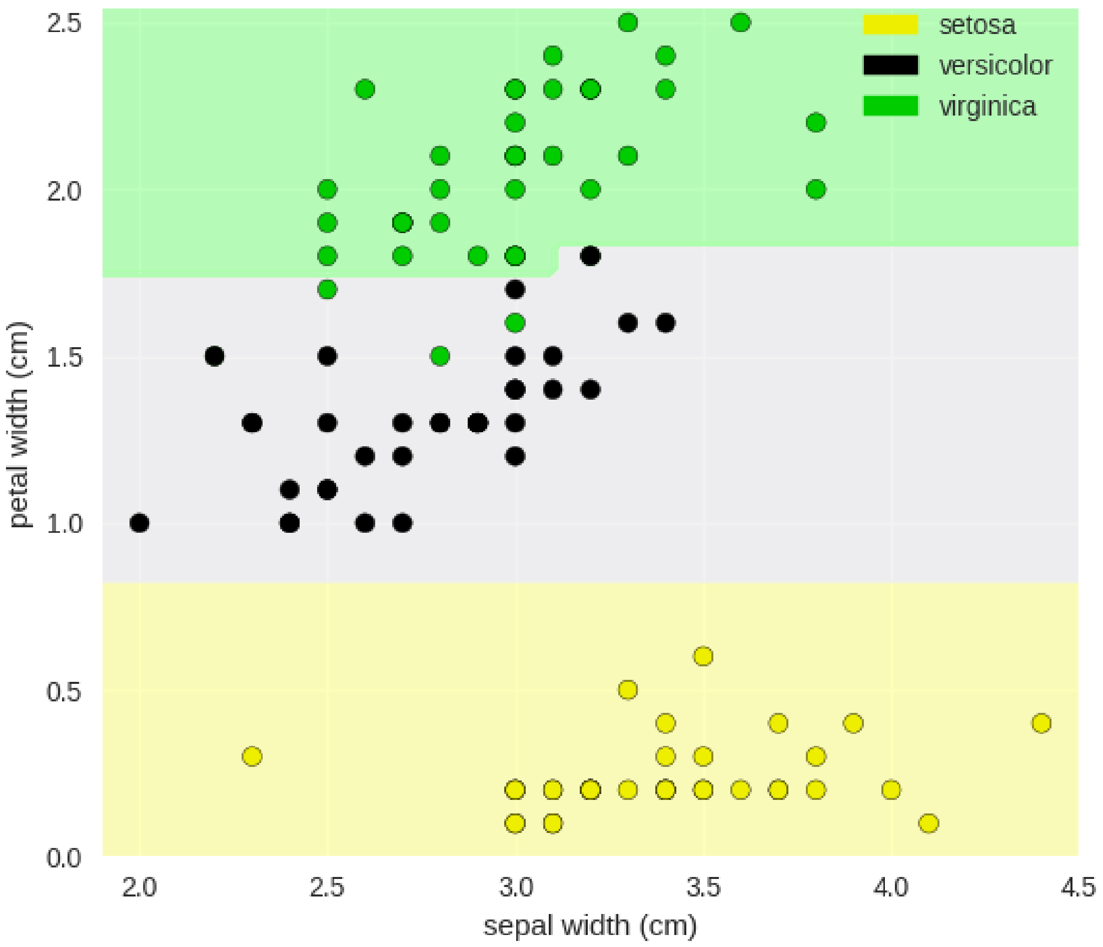


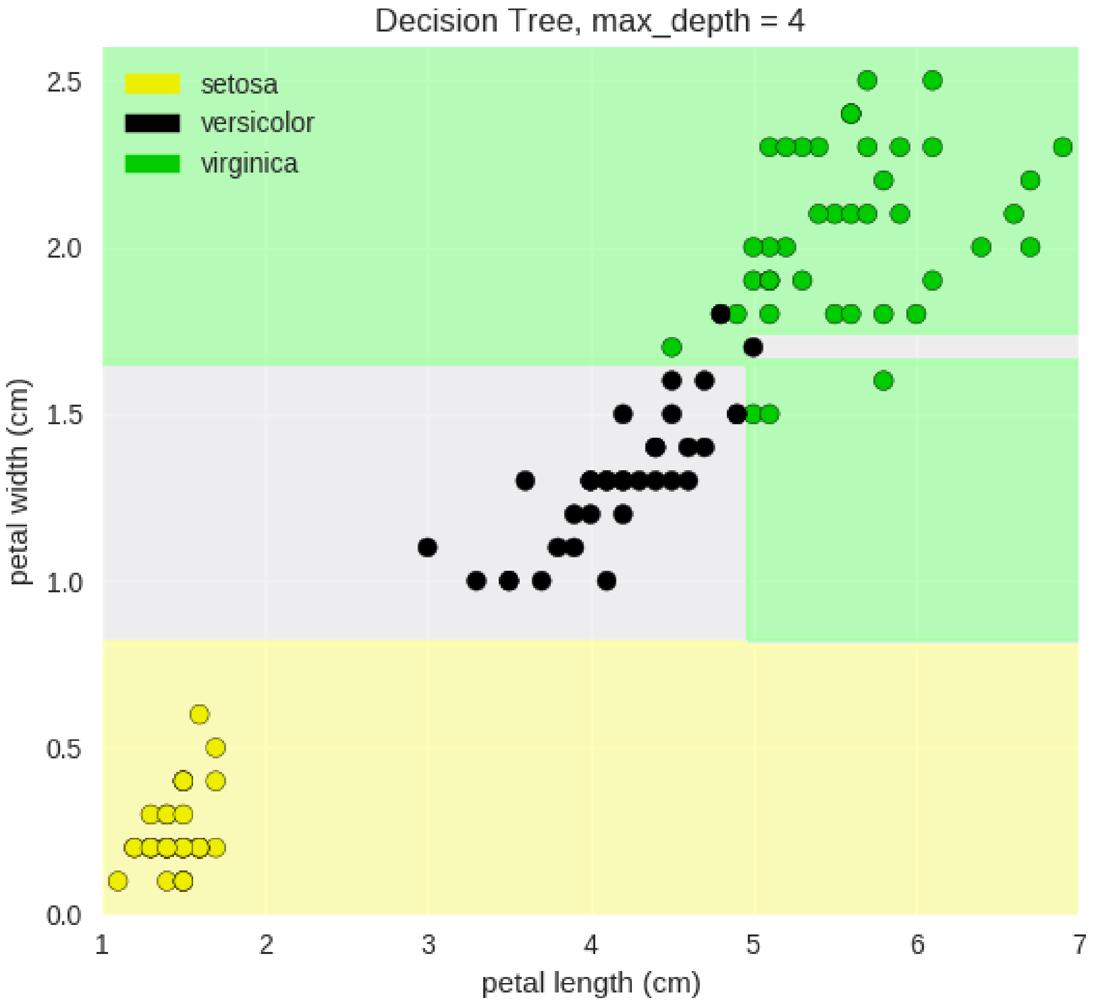
Decision Tree, max_depth = 4



Decision Tree, max_depth = 4




setosa
versicolor
virginica
petal width (cm)
sepal width (cm)
2.5
2.0
1.5
1.0
0.5
0.0
2.0
2.5
3.0
3.5
4.0
4.5

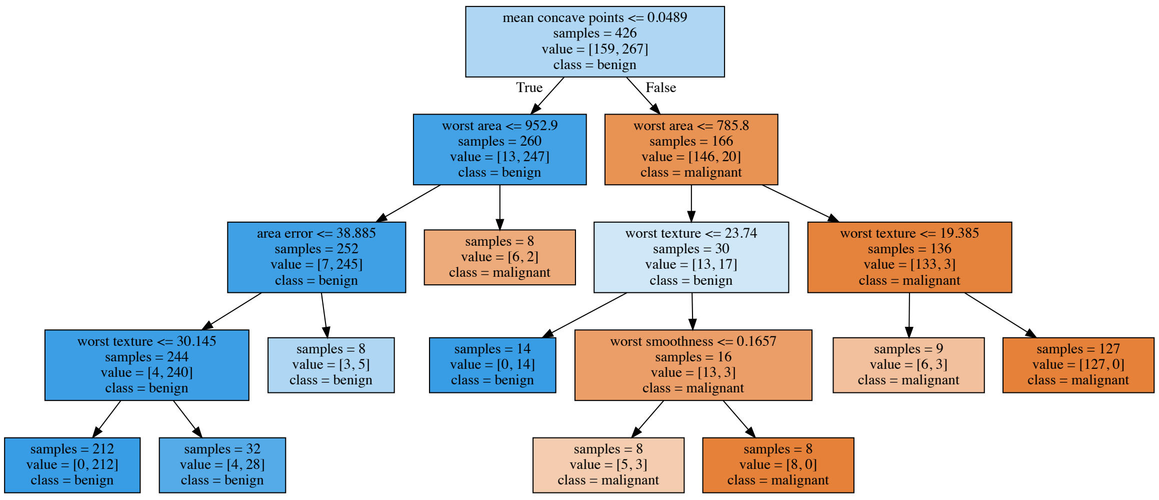

Decision Tree, max_depth = 4
setosa
versicolor
virginica
petal width (cm)
petal length (cm)
2.5
2.0
1.5
1.0
0.5
0.0
1
2
3
4
5
6
7

Decision Trees on a real-world dataset

In [48]:

```
from sklearn.tree import DecisionTreeClassifier
from adspy_shared_utilities import plot_decision_tree
from adspy_shared_utilities import plot_feature_importances

X_train, X_test, y_train, y_test = train_test_split(X_cancer, y_cancer, random_s
tate = 0)

clf = DecisionTreeClassifier(max_depth = 4, min_samples_leaf = 8,
                             random_state = 0).fit(X_train, y_train)

plot_decision_tree(clf, cancer.feature_names, cancer.target_names)
```

Out[48]:

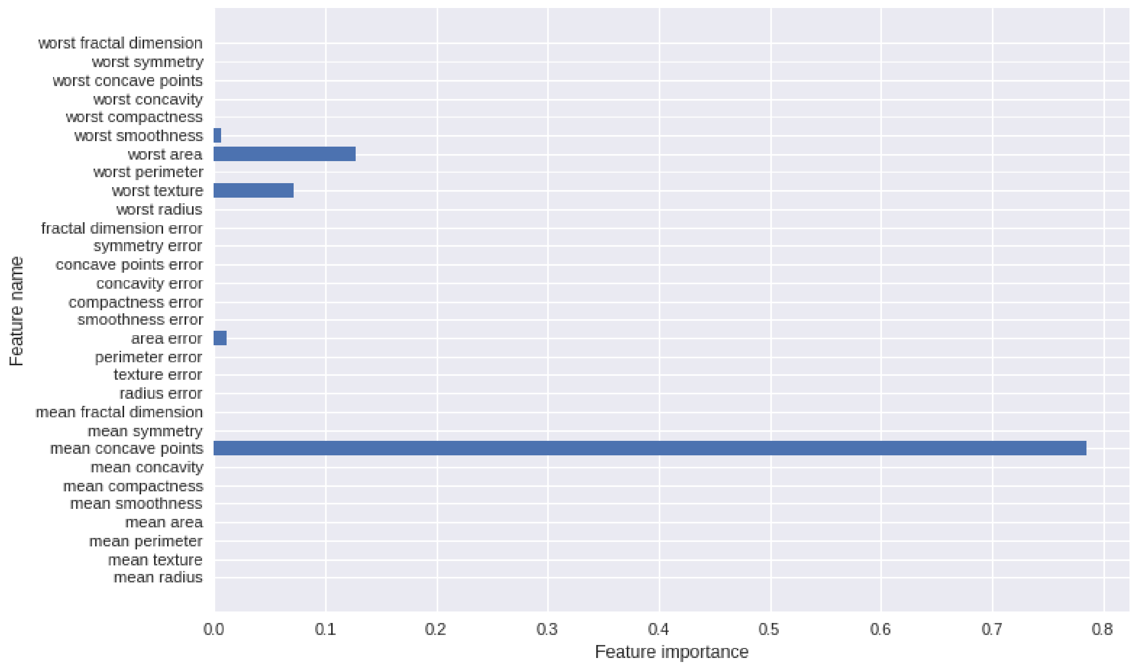


In [49]:

```
print('Breast cancer dataset: decision tree')
print('Accuracy of DT classifier on training set: {:.2f}'
     .format(clf.score(X_train, y_train)))
print('Accuracy of DT classifier on test set: {:.2f}'
     .format(clf.score(X_test, y_test)))

plt.figure(figsize=(10,6),dpi=80)
plot_feature_importances(clf, cancer.feature_names)
plt.tight_layout()

plt.show()
```

```
Breast cancer dataset: decision tree
Accuracy of DT classifier on training set: 0.96
Accuracy of DT classifier on test set: 0.94
```

```
/opt/conda/lib/python3.6/site-packages/matplotlib/pyplot.py:524: Run
timeWarning: More than 20 figures have been opened. Figures created
through the pyplot interface (`matplotlib.pyplot.figure`) are retain
ed until explicitly closed and may consume too much memory. (To cont
rol this warning, see the rcParam `figure.max_open_warning`).
  max_open_warning, RuntimeWarning)
```



In [ ]: