Learning Objectives

- Understand how a number of different supervised learning algorithms learn by estimating their parameters from data to make new predictions.
- Understand the strengths and weaknesses of particular supervised learning methods.

Review of important terms

- Feature representation
- Data instances/samples/ examples (X)
- Target value (y)

fruits

	fruit_label	fruit_name	fruit_subtype	mass	width	height	color_score
0	1	apple	granny_smith	192	8.4	7.3	0.55
1	1	apple	granny_smith	180	8.0	6.8	0.59
2	1	apple	granny_smith	176	7.4	7.2	0.60
3	2	mandarin	mandarin	86	6.2	4.7	0.80
4	2	mandarin	mandarin	84	6.0	4.6	0.79
5	2	mandarin	mandarin	80	5.8	4.3	0.77
6	2	mandarin	mandarin	80	5.9	4.3	0.81
7	2	mandarin	mandarin	76	5.8	4.0	0.81
8	1	apple	braeburn	178	7.1	7.8	0.92
9	1	apple	braeburn	172	7.4	7.0	0.89
10	1	apple	braeburn	166	6.9	7.3	0.93
11	1	apple	braeburn	172	7.1	7.6	0.92
12	1	apple	braeburn	154	7.0	7.1	0.88
13	1	apple	golden_delicious	164	7.3	7.7	0.70
14	1	apple	golden_delicious	152	7.6	7.3	0.69
15	1	apple	golden_delicious	156	7.7	7.1	0.69
16	1	apple	golden_delicious	156	7.6	7.5	0.67
17	1	apple	golden_delicious	168	7.5	7.6	0.73
18	1	apple	cripps_pink	162	7.5	7.1	0.83

Review of important terms

- Training and test sets
- Model/Estimator
  - *Model fitting produces a 'trained model'.*
  - *Training is the process of estimating model parameters.*
- Evaluation method

```
%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

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

X = fruits[['height', 'width', 'mass', 'color_score']]
y = fruits['fruit_label']

X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)

knn = KNeighborsClassifier(n_neighbors = 5)
knn.fit(X_train, y_train)
print("Accuracy of K-NN classifier on test set: ", knn.score(X_test, y_test))

example_fruit = [[5.5, 2.2, 10, 0.70]]
print("Predicted fruit type for ", example_fruit, " is ", knn.predict(example_fruit))
```

Review of important terms

- Training and test sets
- Model/Estimator
  - *Model fitting produces a 'trained model'.*
  - *Training is the process of estimating model parameters.*
- Evaluation method

```
%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

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

X = fruits[['height', 'width', 'mass', 'color_score']]
y = fruits['fruit_label']

X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)

knn = KNeighborsClassifier(n_neighbors = 5)
knn.fit(X_train, y_train)
print("Accuracy of K-NN classifier on test set: ", knn.score(X_test, y_test))

example_fruit = [[5.5, 2.2, 10, 0.70]]
print("Predicted fruit type for ", example_fruit, " is ", knn.predict(example_fruit))
```

Review of important terms

- Training and test sets
- Model/Estimator
  - *Model fitting produces a 'trained model'.*
  - *Training is the process of estimating model parameters.*
- Evaluation method

```
%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

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

X = fruits[['height', 'width', 'mass', 'color_score']]
y = fruits['fruit_label']

X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)

knn = KNeighborsClassifier(n_neighbors = 5)
knn.fit(X_train, y_train)
print("Accuracy of K-NN classifier on test set: ", knn.score(X_test, y_test))

example_fruit = [[5.5, 2.2, 10, 0.70]]
print("Predicted fruit type for ", example_fruit, " is ", knn.predict(example_fruit))
```

Classification and Regression

- Both classification and regression take a set of training instances and learn a mapping to a target value.
- For classification, the target value is a discrete class value
  - *Binary: target value is 0 (negative class) or 1 (positive class)*
    - *e.g. detecting a fraudulent credit card transaction*

Binary classification: credit card fraud detection

Transaction information

Card Number xxxx xxxx xxxx 0459
Seattle, WA, U.S.A.
Peak Place Market
USD $5494.30
Shoes R Us

User transaction history

Yes / No

Multi-class classification: fruit recognition

Physical attributes

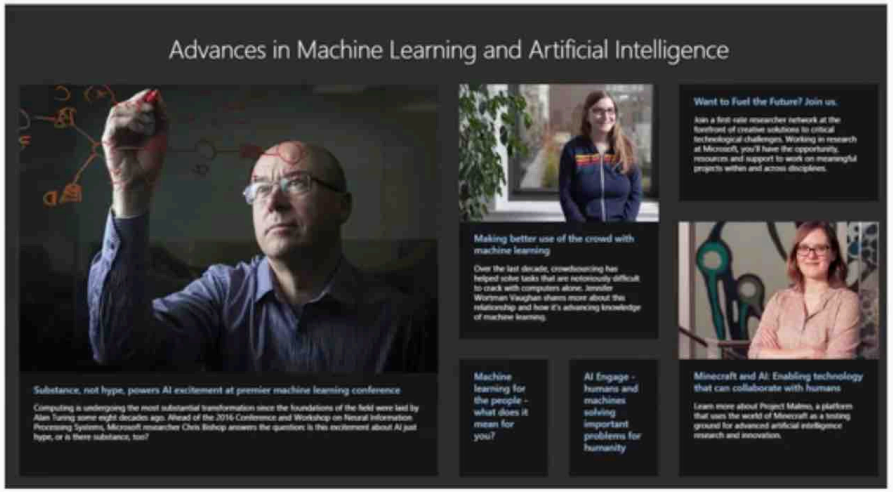
Height
Width
Mass
Color index

Fruit class

- Apple
- Lemon
- Orange
- Mandarin Orange

Multi-label classification: classifying Web pages into multiple topics

research.microsoft.com

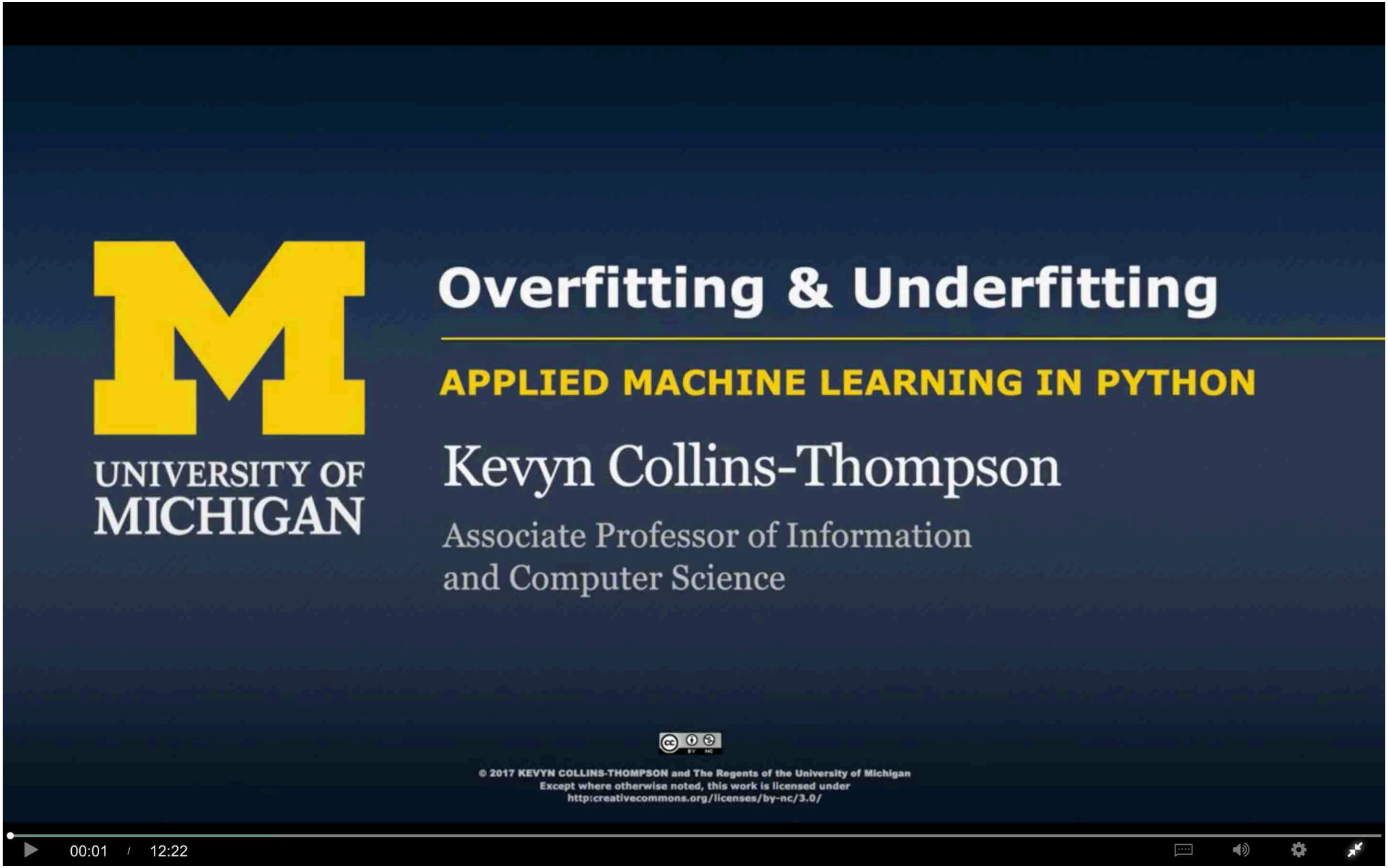


Classes

- 0.83 computers
- 0.20 science
- 0.08 society/issues
- 0.07 reference
- 0.03 reference/education

Classification and Regression

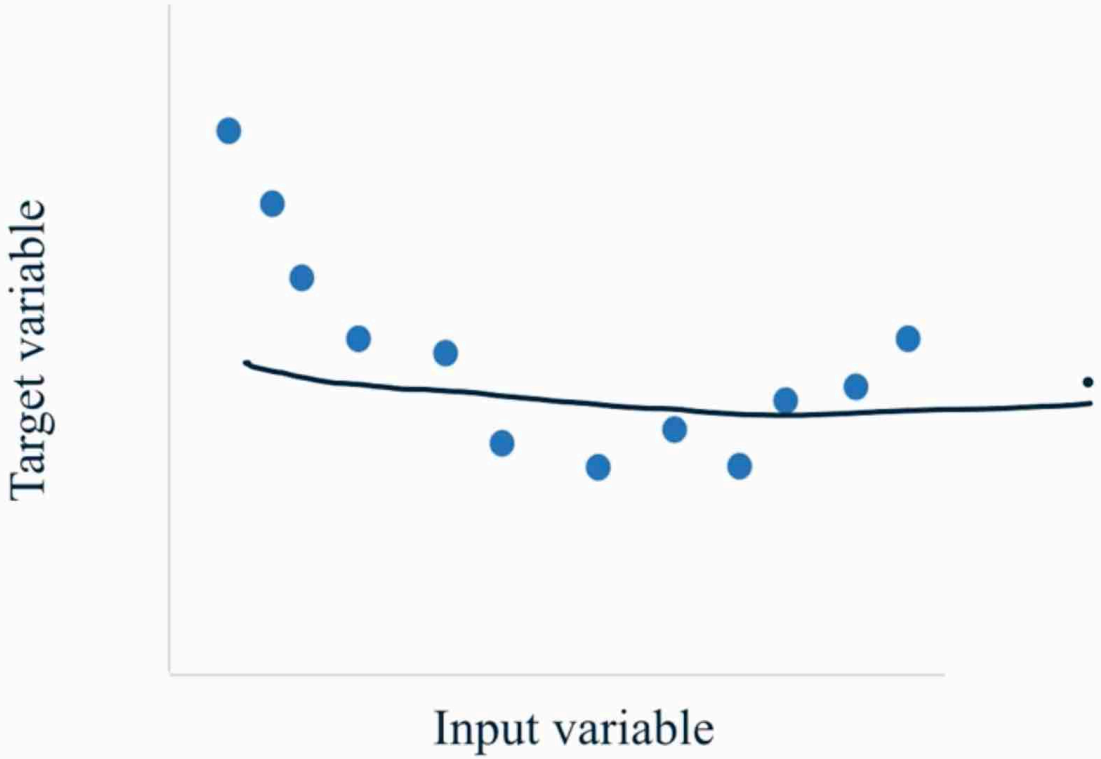
- For regression, the target value is continuous (floating point/real-valued).
  - *e.g. predicting the selling price of a house from its attributes*
- Looking at the target value's type will guide you on what supervised learning method to use.
- Many supervised learning methods have 'flavors' for both classification and regression.


UNIVERSITY OF MICHIGAN
Overfitting & Underfitting
APPLIED MACHINE LEARNING IN PYTHON
Kevyn Collins-Thompson
Associate Professor of Information and Computer Science
© 2017 KEVYN COLLINS-THOMPSON and The Regents of the University of Michigan
Except where otherwise noted, this work is licensed under
http:creativecommons.org/licenses/by-nc/3.0/
00:01 / 12:22

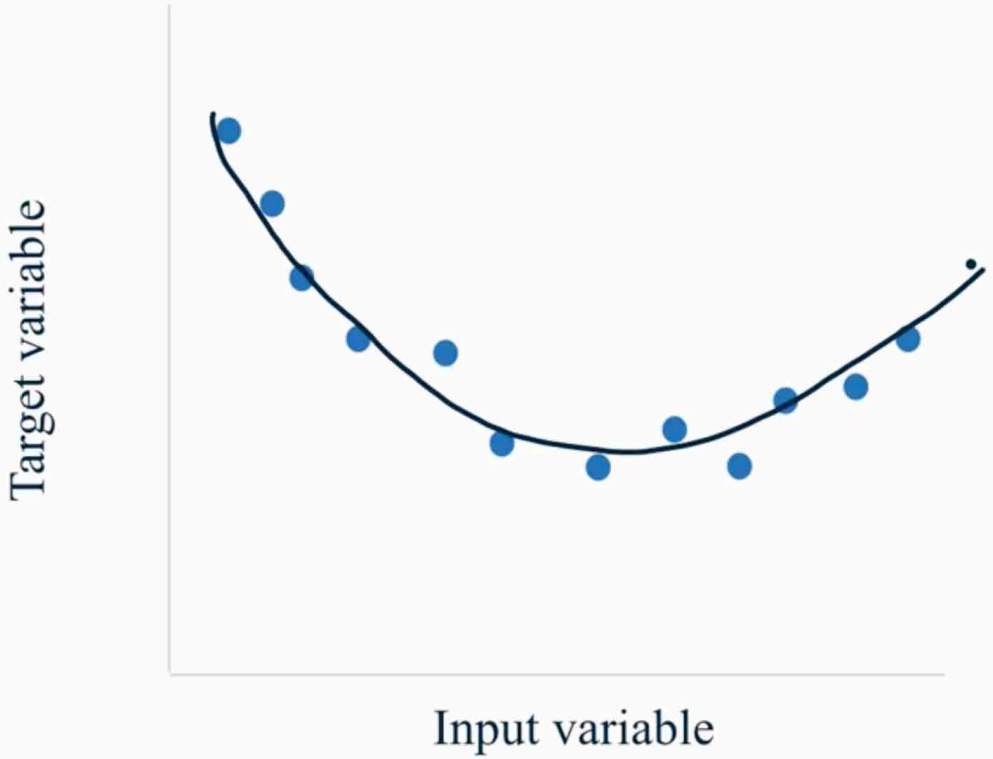
Generalization, Overfitting, and Underfitting

- Generalization ability refers to an algorithm's ability to give accurate predictions for new, previously unseen data.
- Assumptions:
  - *Future unseen data (test set) will have the same properties as the current training sets.*
  - *Thus, models that are accurate on the training set are expected to be accurate on the test set.*
  - *But that may not happen if the trained model is tuned too specifically to the training set.*
- Models that are too complex for the amount of training data available are said to overfit and are not likely to generalize well to new examples.
- Models that are too simple, that don't even do well on the training data, are said to underfit and also not likely to generalize well.

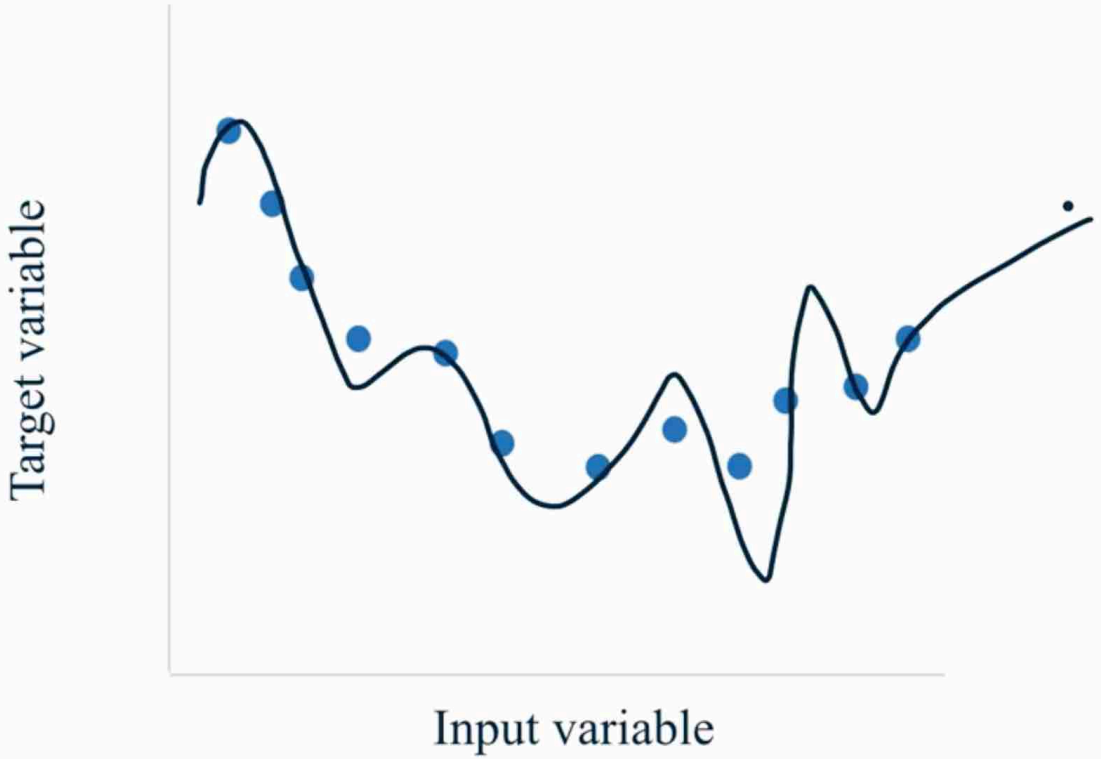
Overfitting in regression



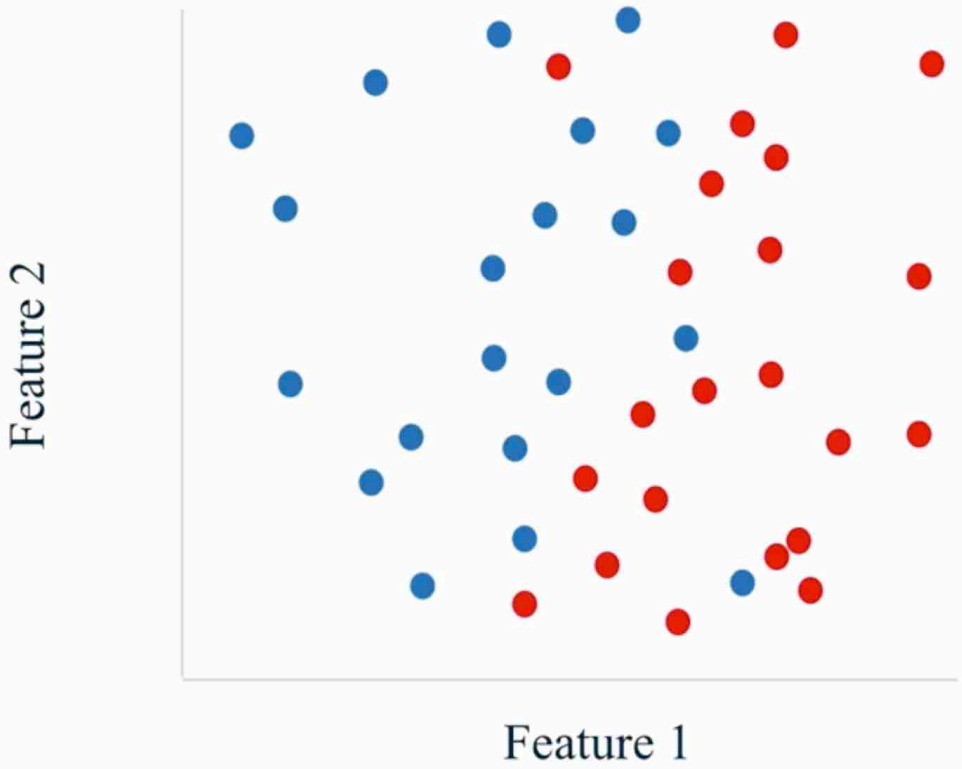
Overfitting in regression



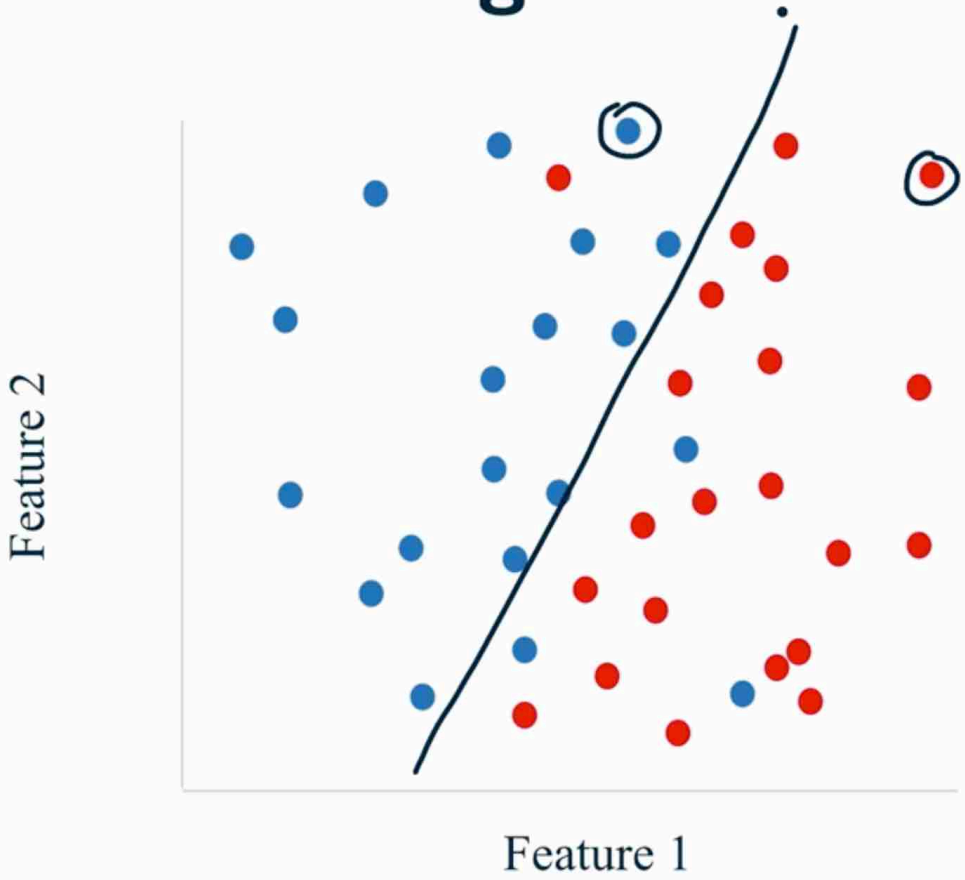
Overfitting in regression



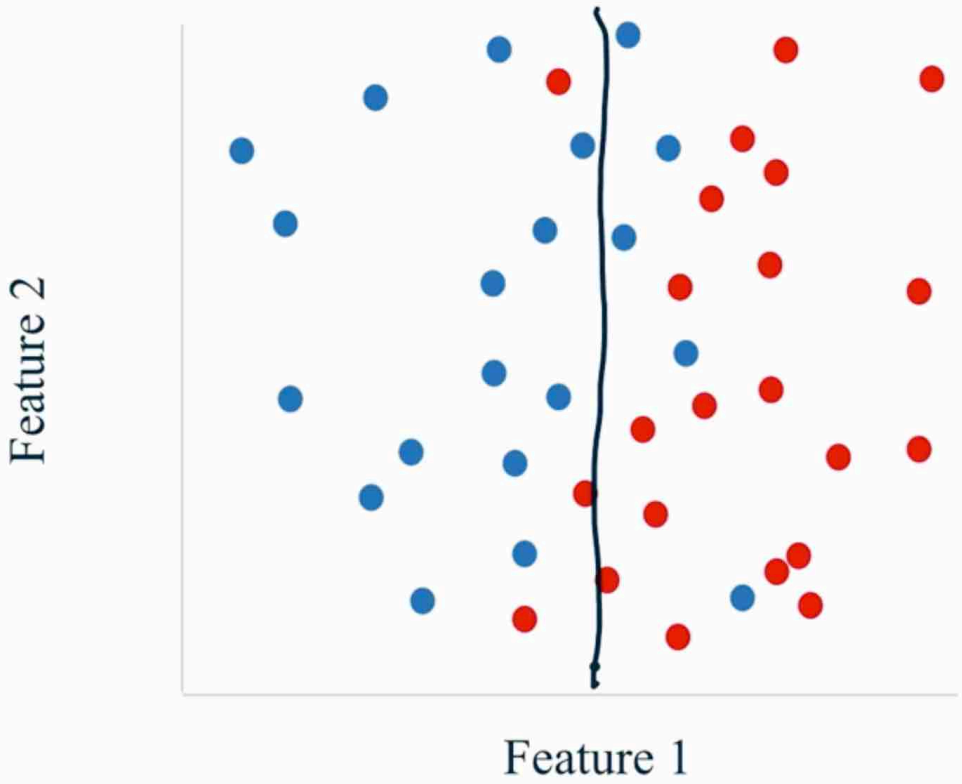
Overfitting in classification



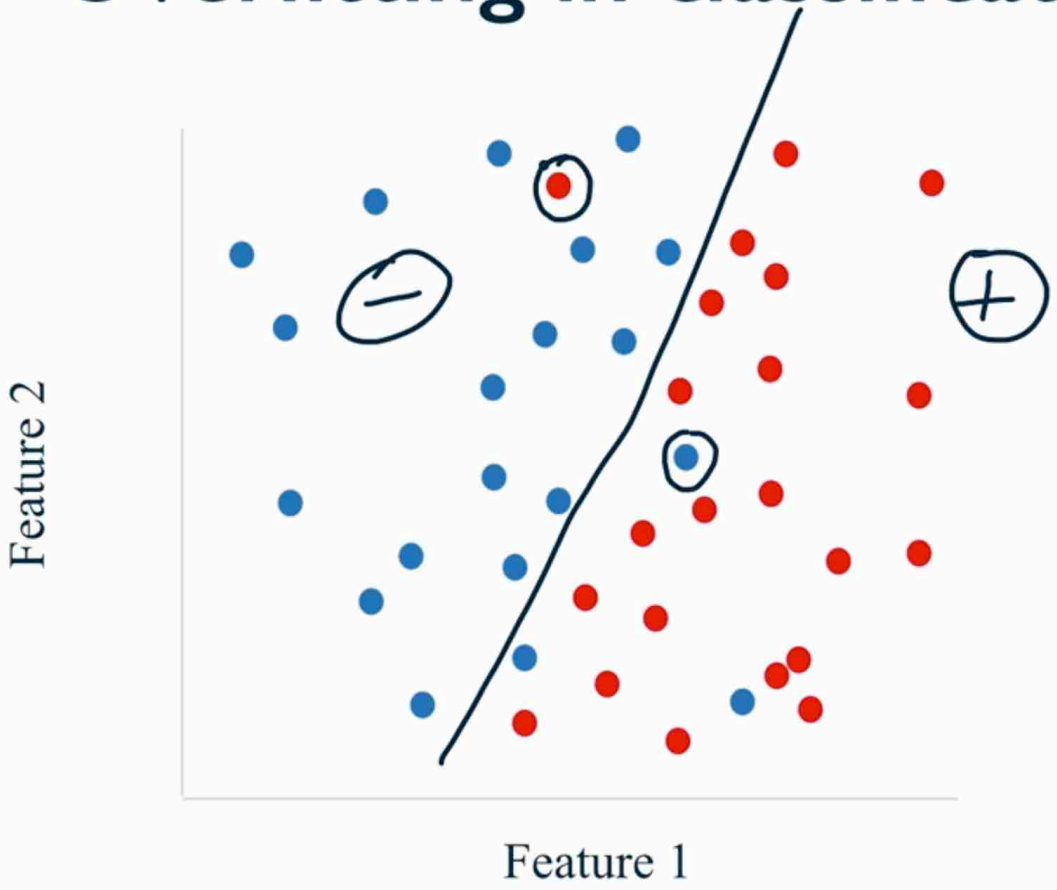
Overfitting in classification



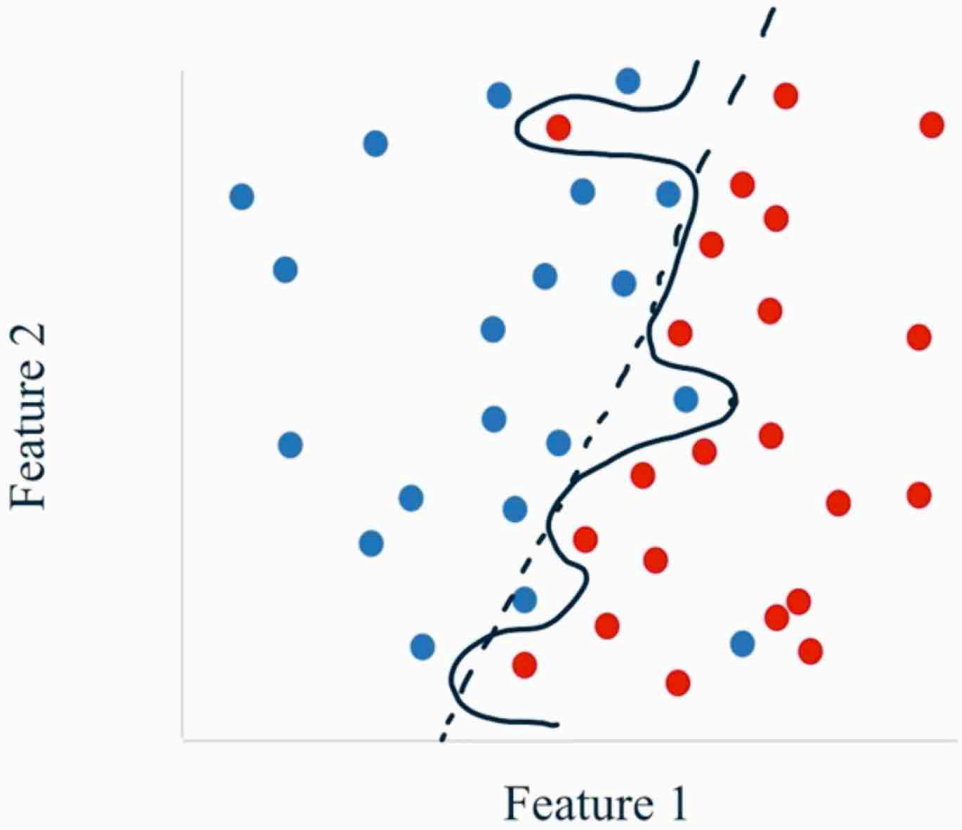
Overfitting in classification



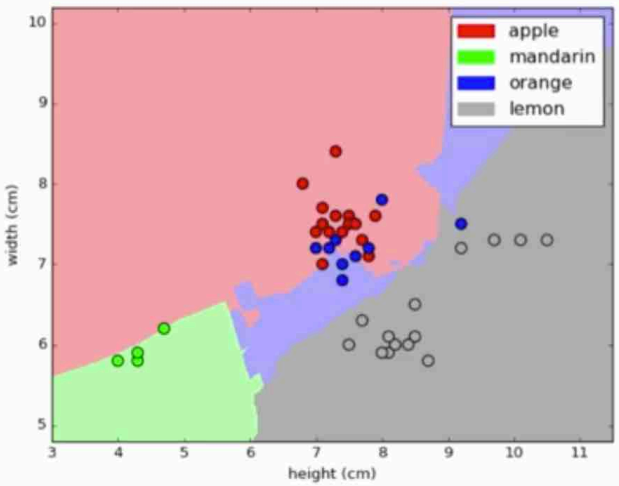
Overfitting in classification



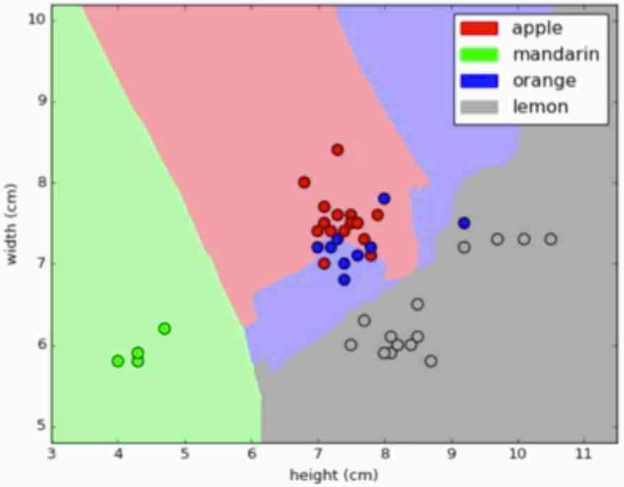
Overfitting in classification



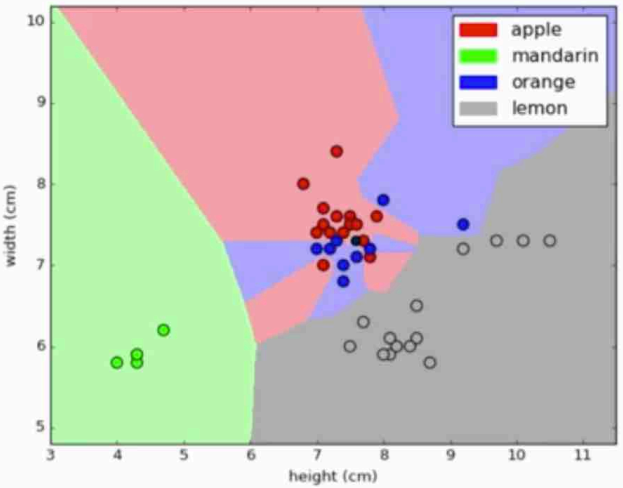
Overfitting with k-NN classifiers



K=10

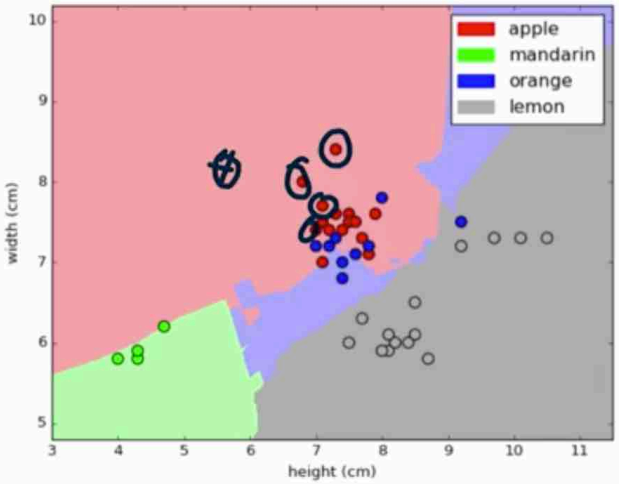


K=5

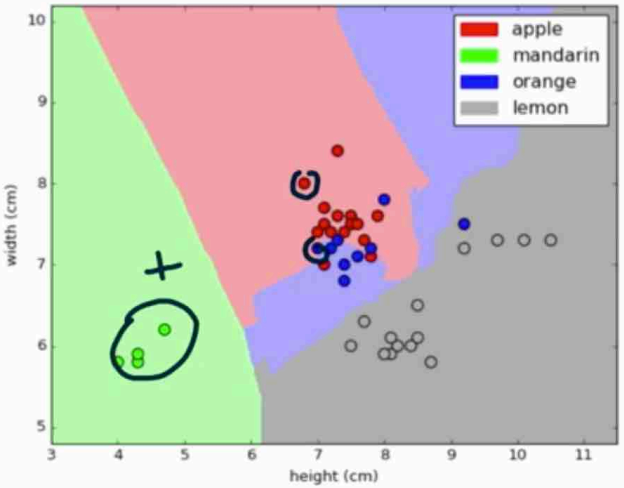


K=1

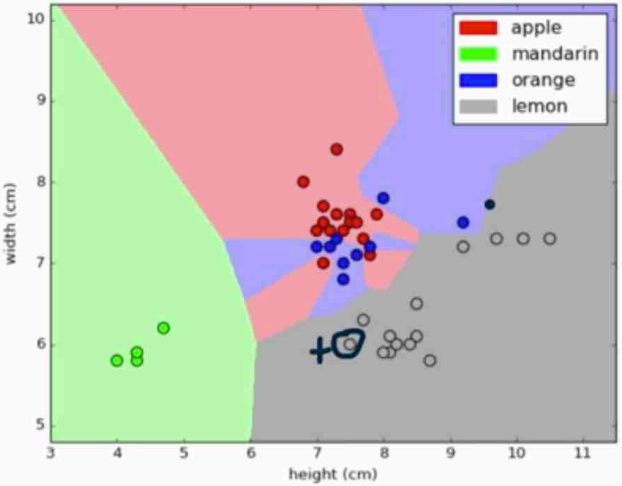
Overfitting with k-NN classifiers



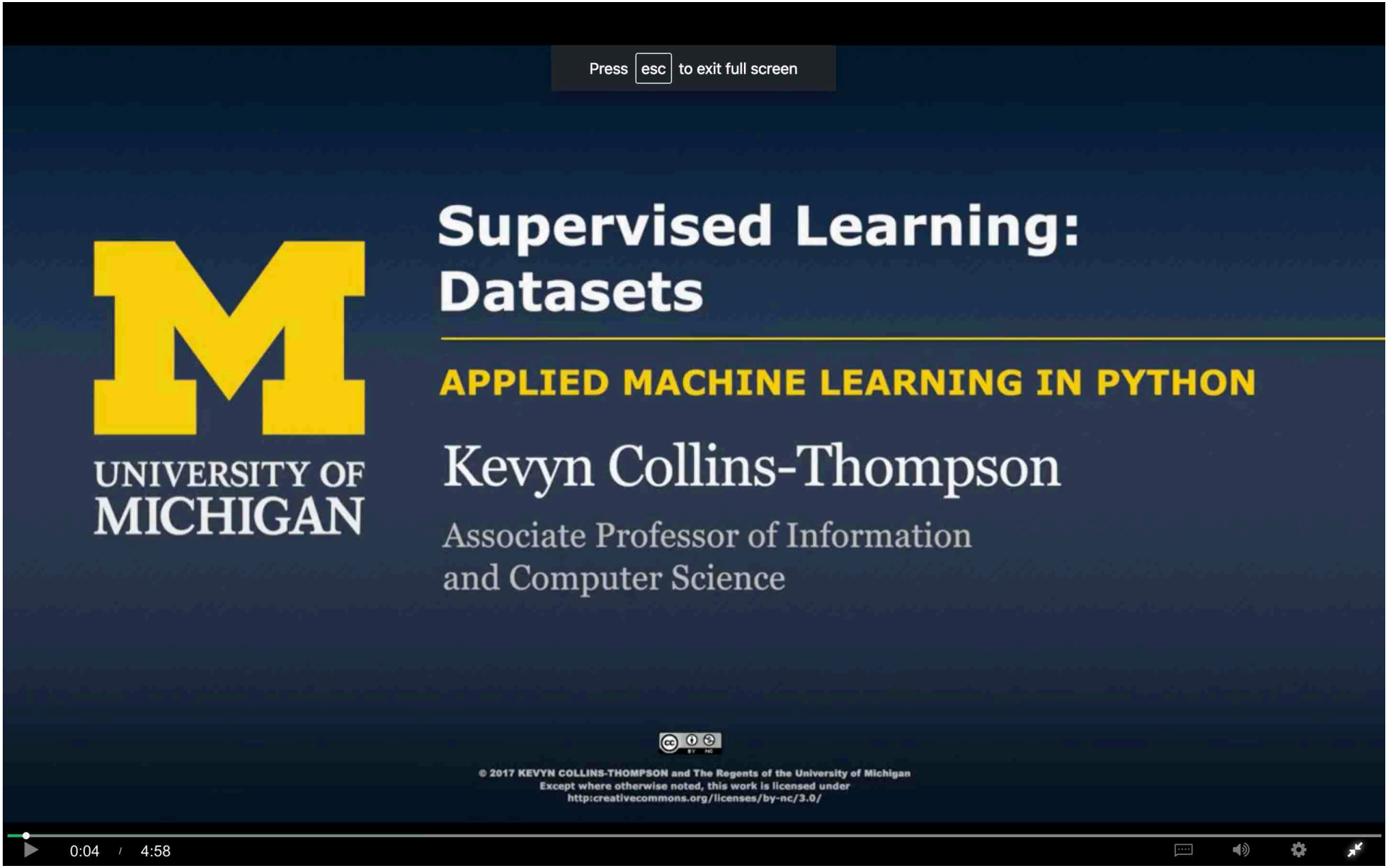
K=10



K=5

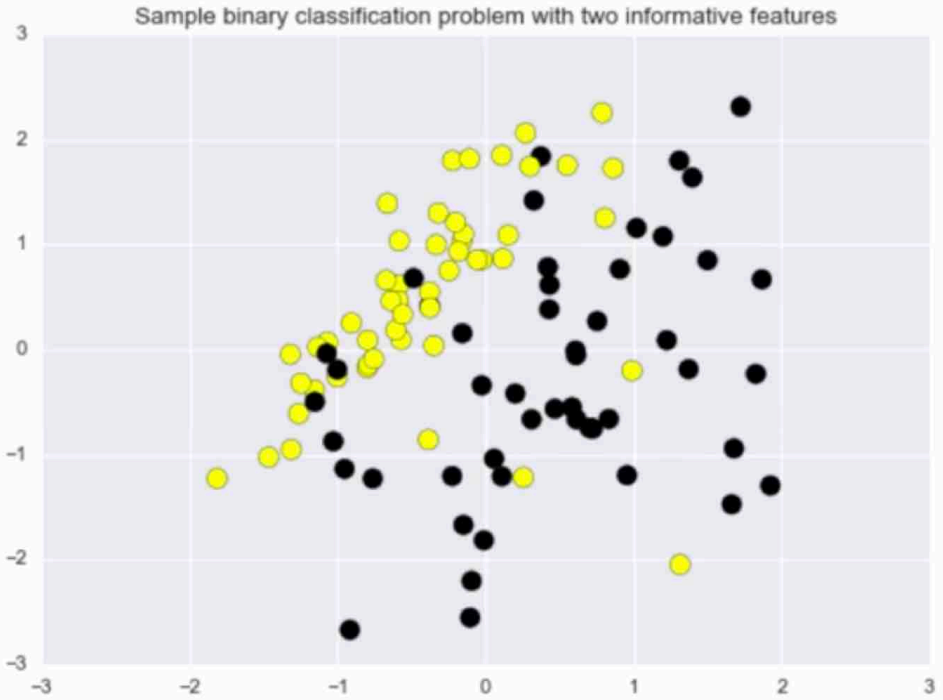


K=1

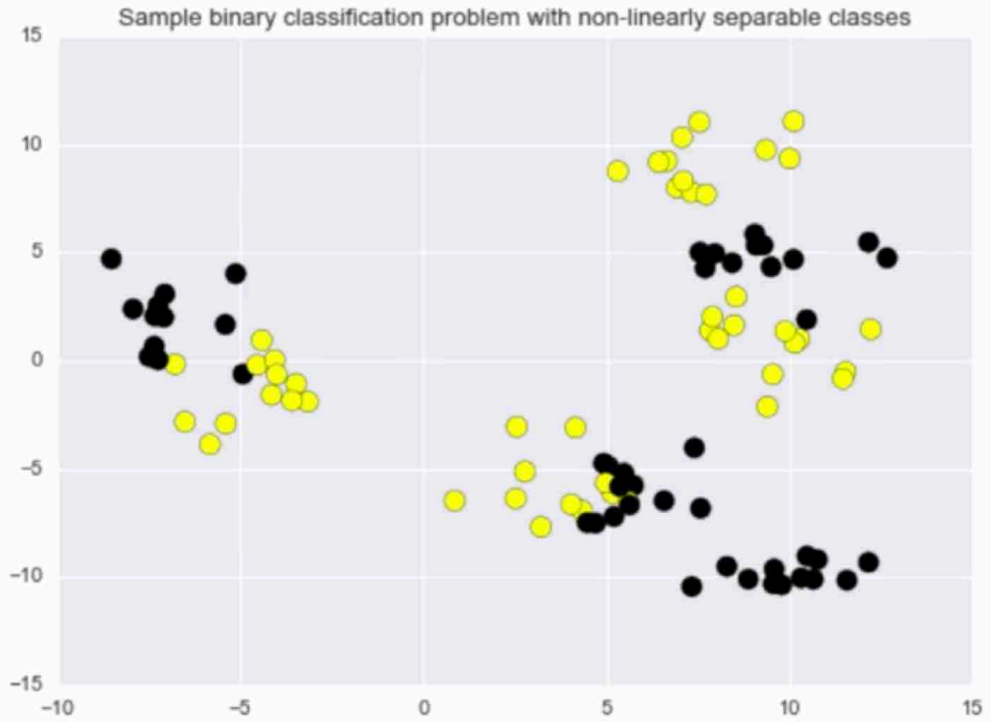

Press esc to exit full screen
Supervised Learning: Datasets
APPLIED MACHINE LEARNING IN PYTHON
UNIVERSITY OF MICHIGAN
Kevyn Collins-Thompson
Associate Professor of Information and Computer Science
© 2017 KEVYN COLLINS-THOMPSON and The Regents of the University of Michigan
Except where otherwise noted, this work is licensed under
http:creativecommons.org/licenses/by-nc/3.0/
0:04 / 4:58

Simple regression dataset

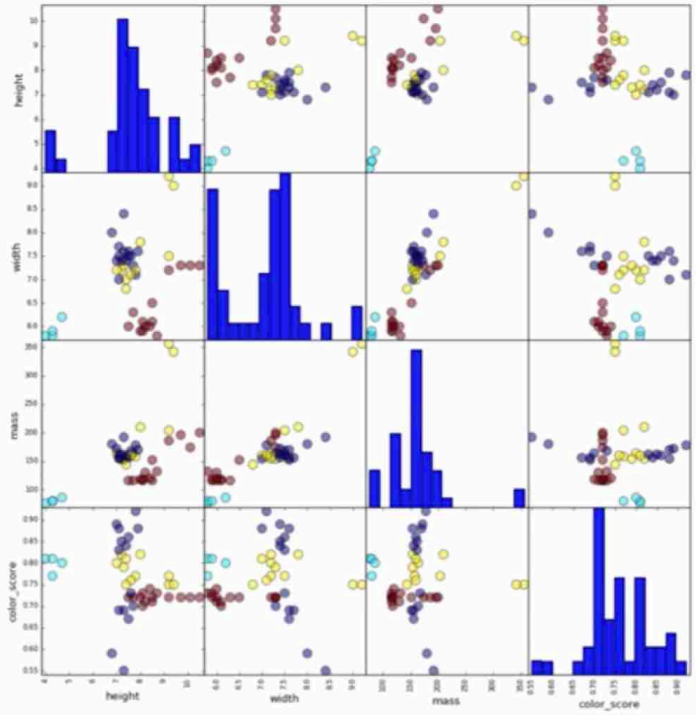
Simple binary classification dataset



Complex binary classification dataset



Fruits multi-class classification dataset



Features

- `width`
- `height`
- `mass`
- `color_index`

Classes

0: apple

1: mandarin orange

2: orange

3: lemon

Communities and Crime dataset

	population	householdsize	agePct12t21	agePct12t29	agePct16t24	agePct65up	numbUrban	pctUrban	medIncome	pctWWage	...	MedRentF
0	11980	3.10	12.47	21.44	10.93	11.33	11980	100.0	75122	89.24	...	23.8
1	23123	2.82	11.01	21.30	10.48	17.18	23123	100.0	47917	78.99	...	27.6
2	29344	2.43	11.36	25.88	11.01	10.28	29344	100.0	35669	82.00	...	24.1
3	16656	2.40	12.55	25.20	12.19	17.57	0	0.0	20580	68.15	...	28.7
5	140494	2.45	18.09	32.89	20.04	13.26	140494	100.0	21577	75.78	...	26.4
6	28700	2.60	11.17	27.41	12.76	14.42	28700	100.0	42805	79.47	...	24.4
7	59459	2.45	15.31	27.93	14.78	14.60	59449	100.0	23221	71.60	...	26.3
8	74111	2.46	16.64	35.16	20.33	8.58	74115	100.0	25326	83.69	...	25.2
9	103590	2.62	19.88	34.55	21.62	13.12	103590	100.0	17852	74.20	...	29.6
10	31601	2.54	15.73	28.57	15.16	14.26	31596	100.0	24763	73.92	...	23.8

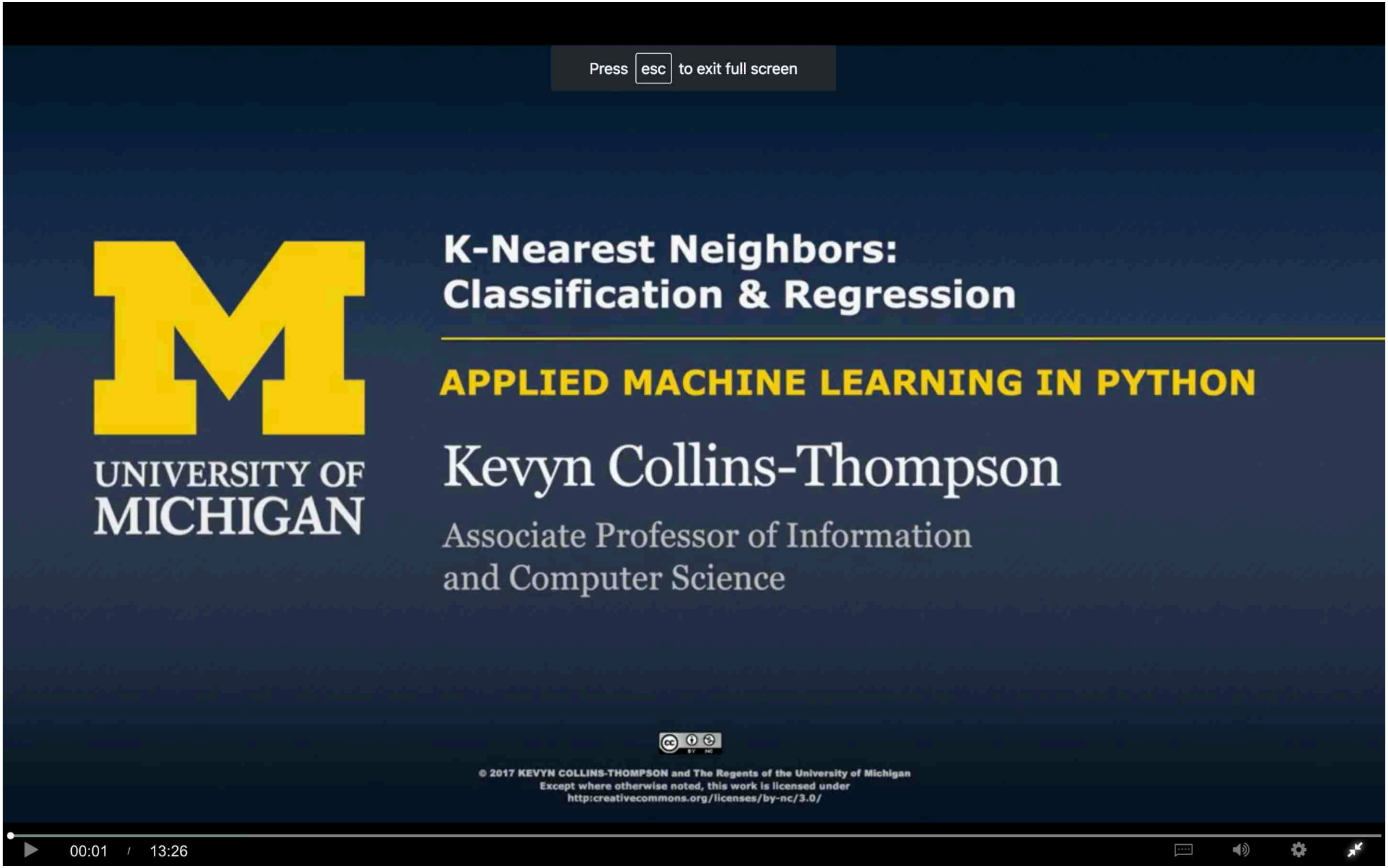
Input features:
socio-economic data by location from U.S. Census

Target variable:
Per capita violent crimes

Derived from the original UCI dataset at:

https://archive.ics.uci.edu/ml/datasets/Communities+and+Crime+Unnormalized

```
from adspy_shared_utilities import load_crime_dataset
crime = load_crime_dataset()
```

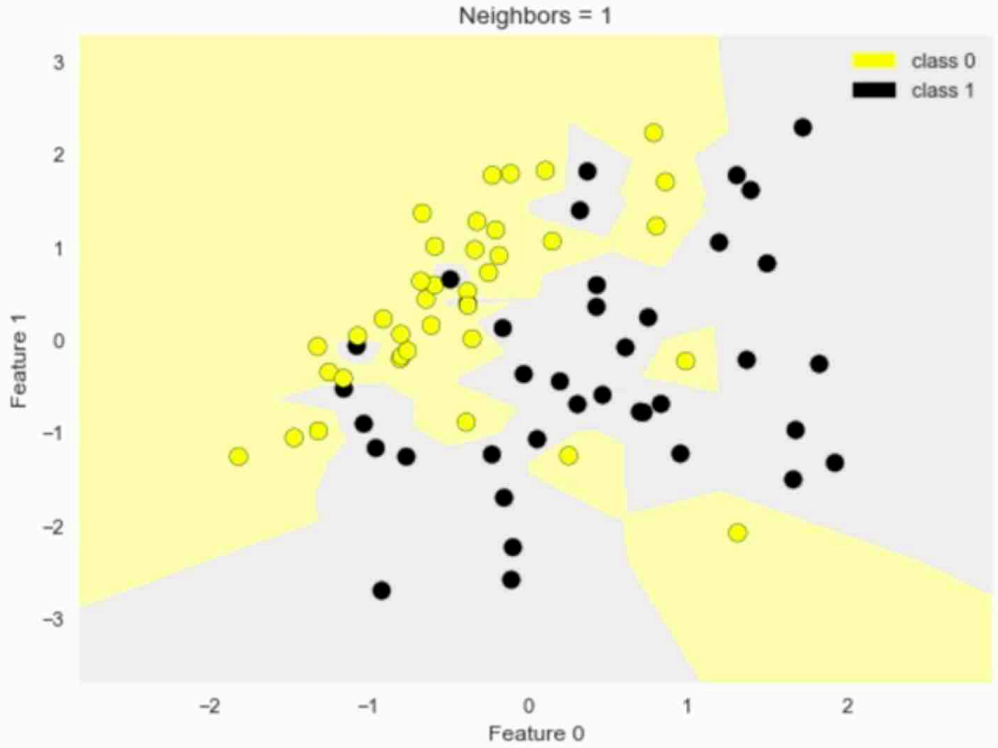

Press esc to exit full screen
UNIVERSITY OF MICHIGAN
K-Nearest Neighbors: Classification & Regression
APPLIED MACHINE LEARNING IN PYTHON
Kevyn Collins-Thompson
Associate Professor of Information and Computer Science
© 2017 KEVYN COLLINS-THOMPSON and The Regents of the University of Michigan
Except where otherwise noted, this work is licensed under
http:creativecommons.org/licenses/by-nc/3.0/
00:01 / 13:26

The k-Nearest Neighbor (k-NN) Classifier Algorithm

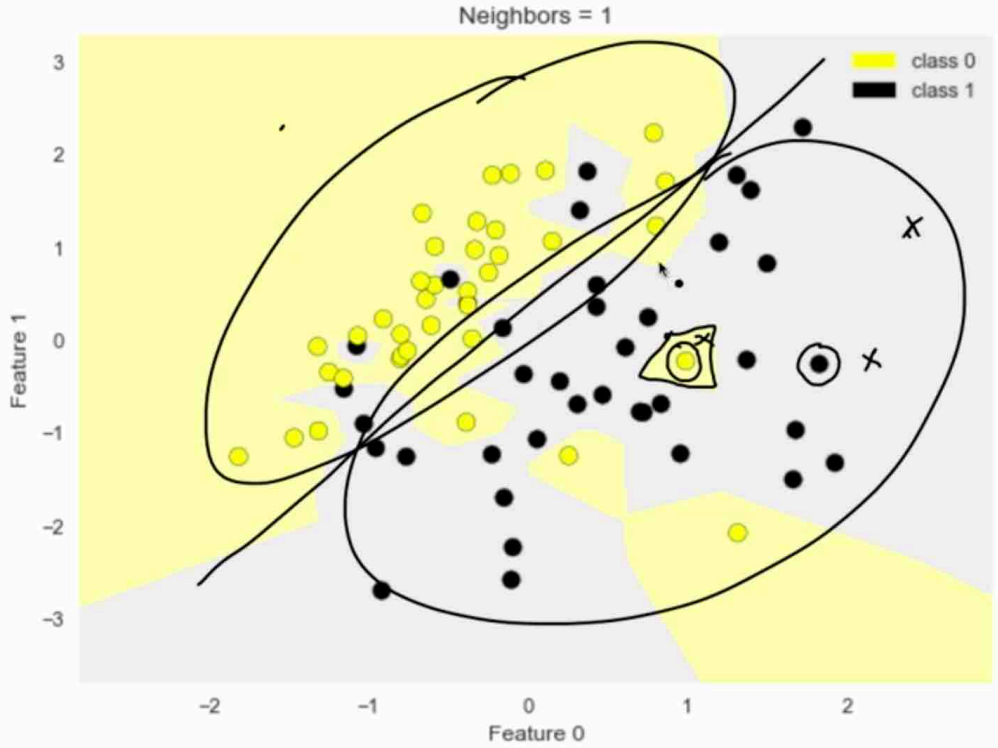
Given a training set X_train with labels y_train, and given a new instance x_test to be classified:

1. Find the most similar instances (let's call them X_NN) to x_test that are in X_train.
2. Get the labels y_NN for the instances in X_NN
3. Predict the label for x_test by combining the labels y_NN e.g. simple majority vote

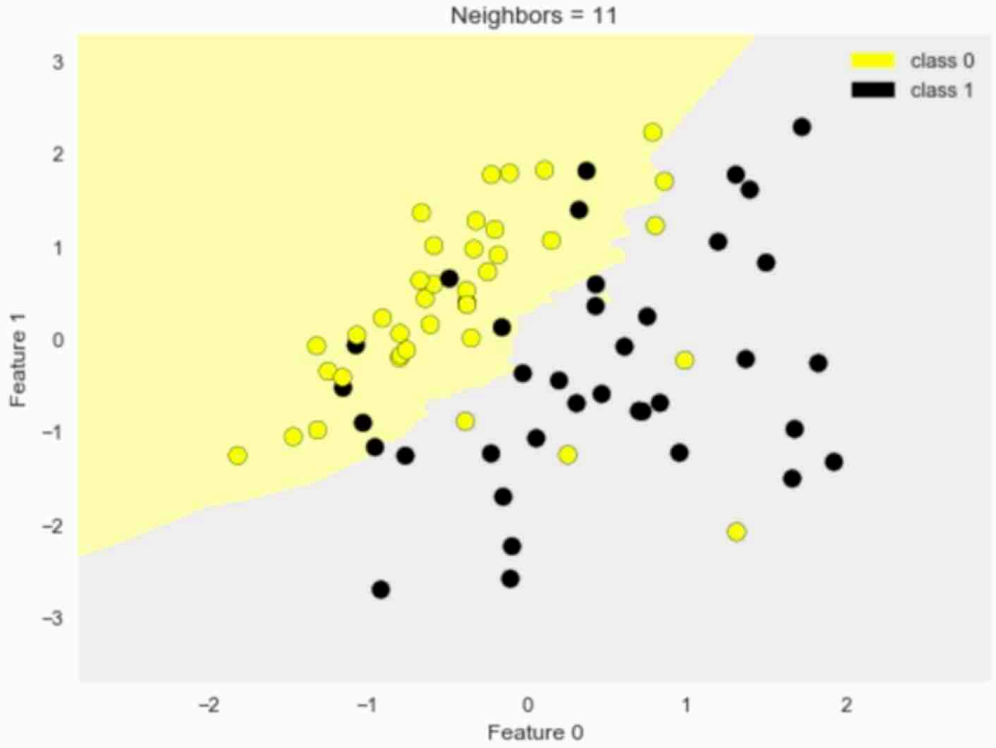
Nearest neighbors classification (k=1)



Nearest neighbors classification (k=1)



Nearest neighbors classification (k=11)



K-Nearest Neighbors

Classification

In [*]:
```
from adspy_shared_utilities import plot_two_class_knn

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)
```

In [ ]:
```

```

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

Figure 5

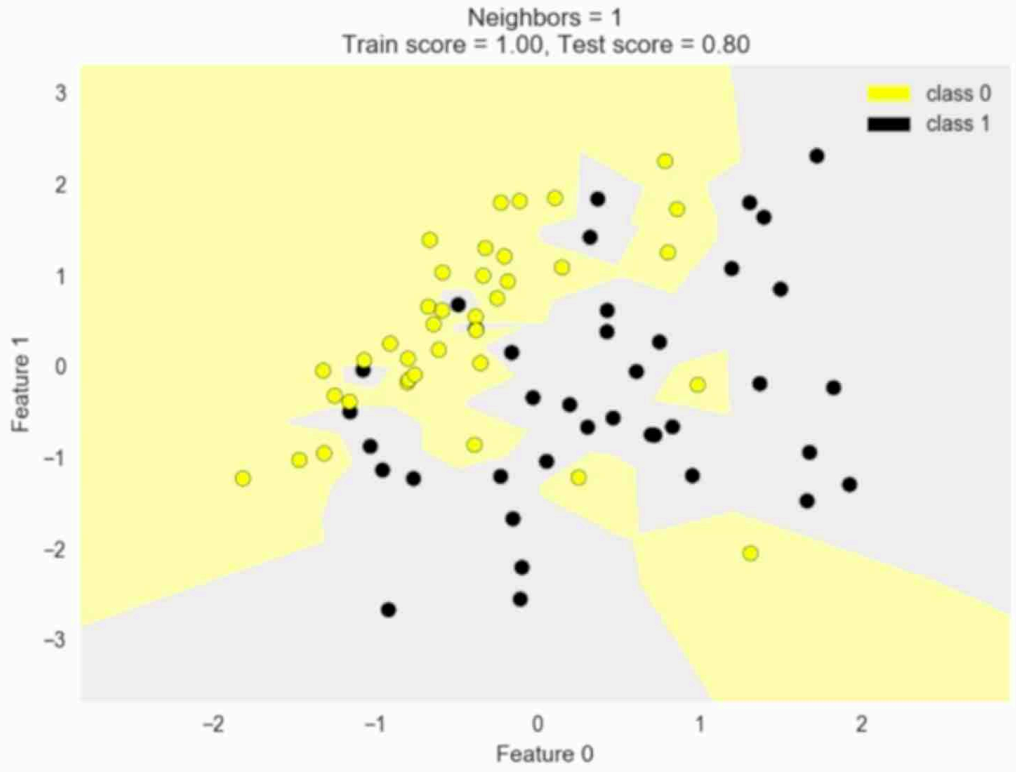


Figure 6

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

Figure 5

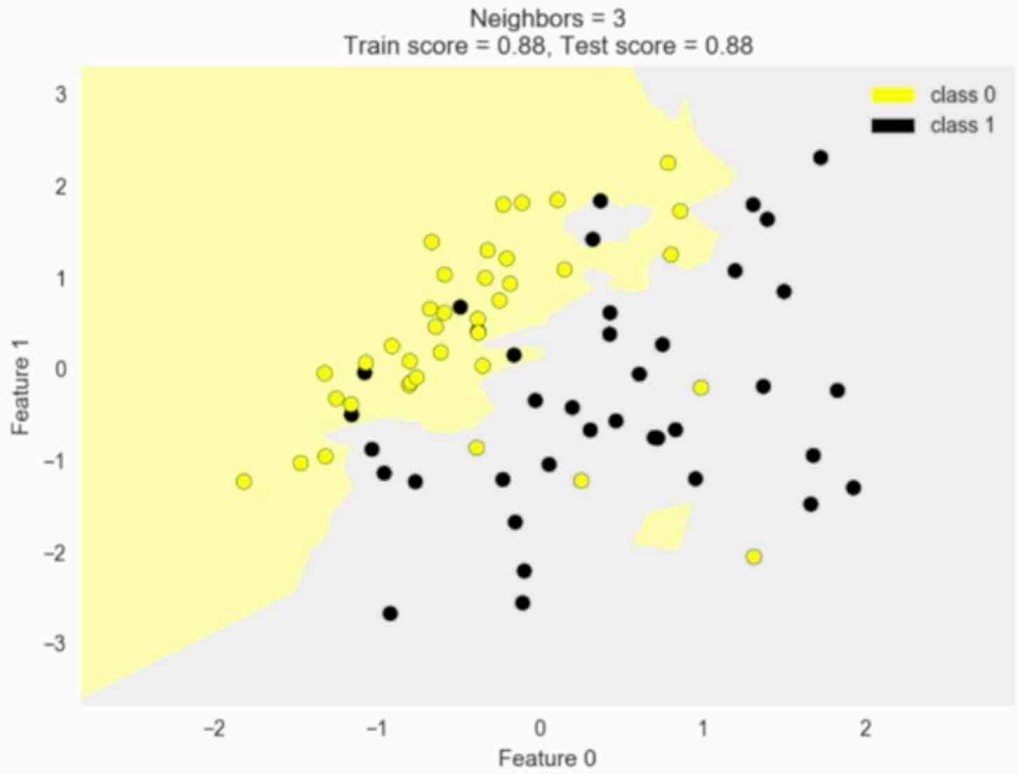


Figure 6

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

Figure 5

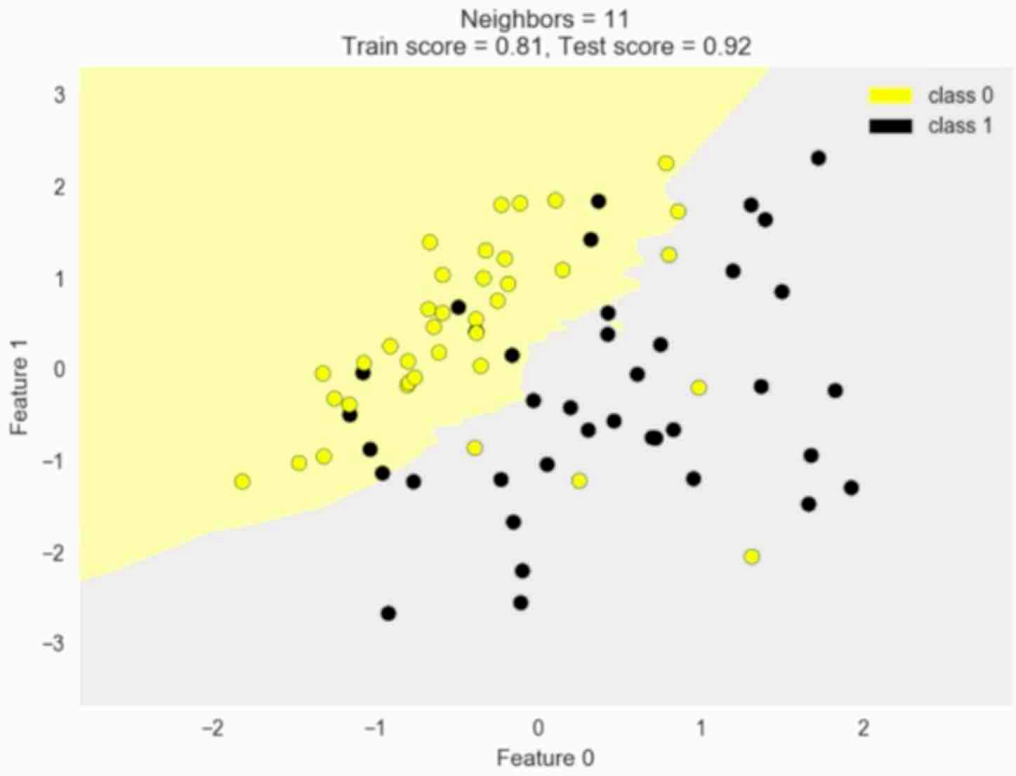
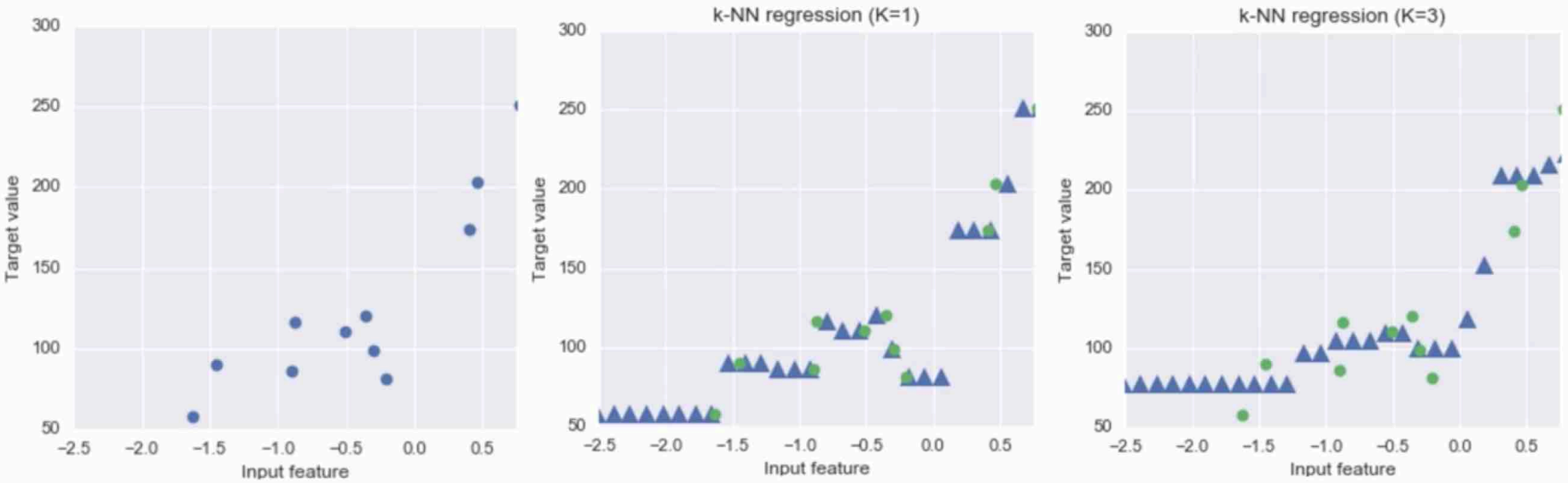
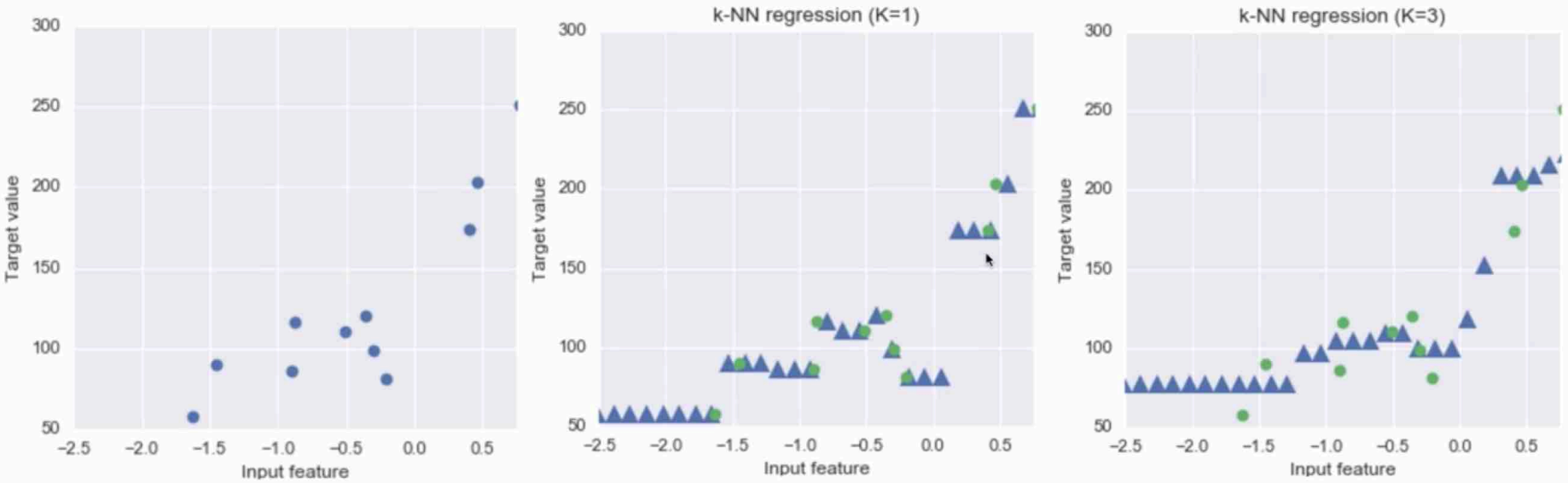


Figure 6

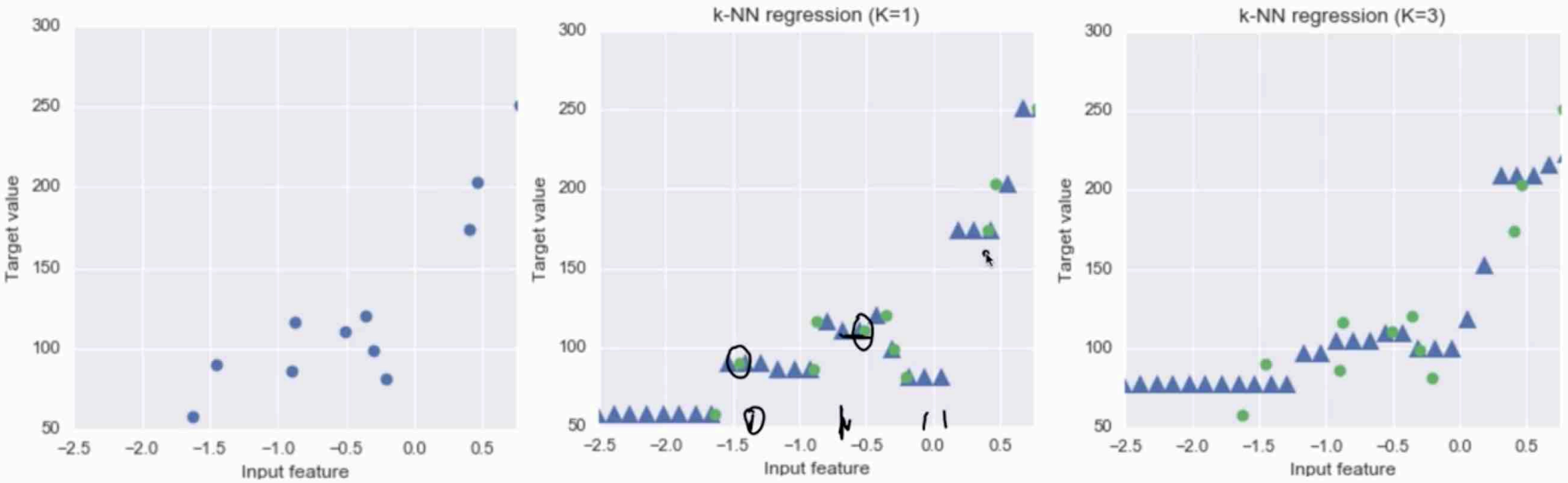
k-Nearest neighbors regression



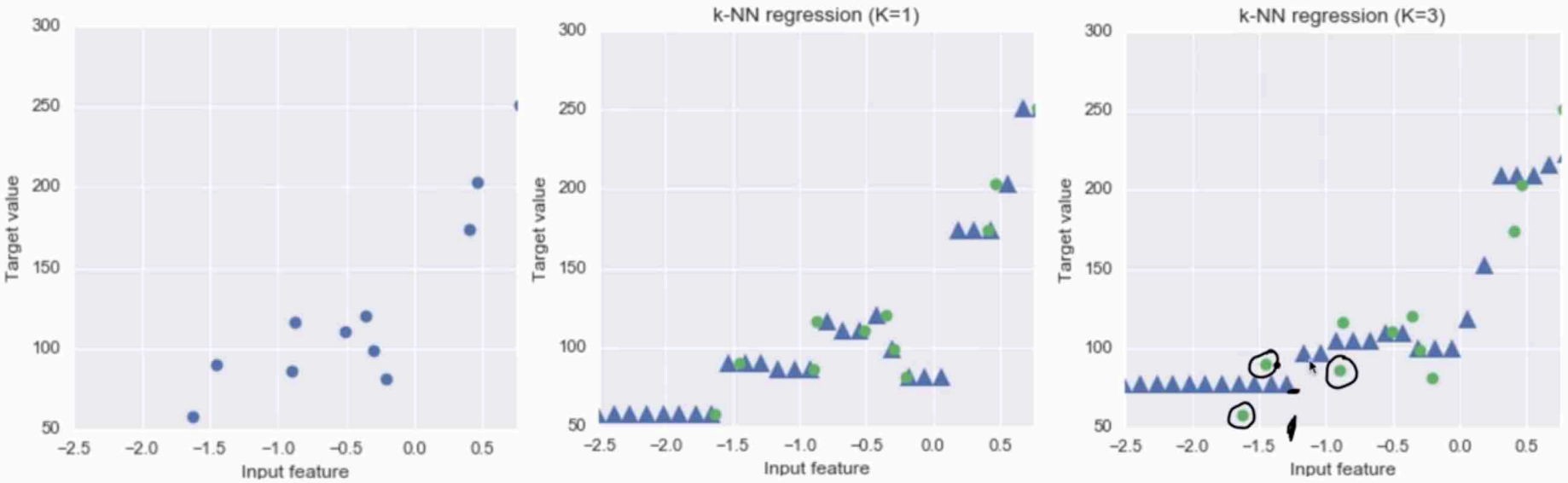
k-Nearest neighbors regression



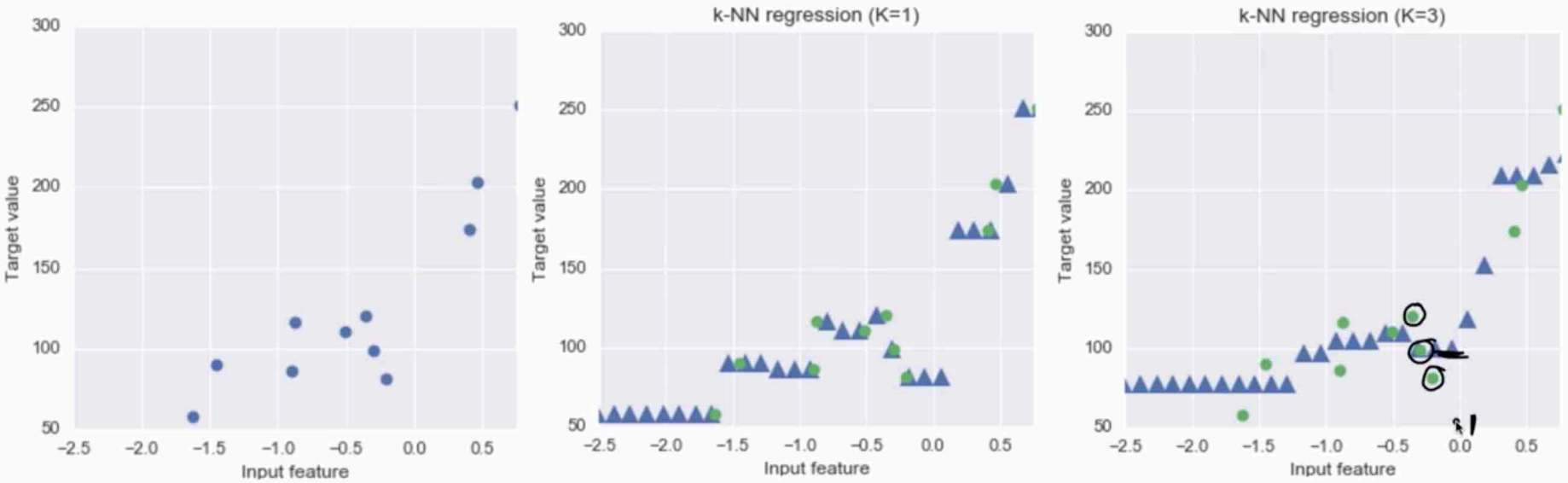
k-Nearest neighbors regression



k-Nearest neighbors regression



k-Nearest neighbors regression



Regression

In [4]:
```
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  208.26
  102.1   191.32  134.5   228.32  148.36  159.17  113.47  144.04  199.23
  143.19  166.51  231.71  208.26  128.02  123.14  141.91]
R-squared test score: 0.425
```

In [ ]:

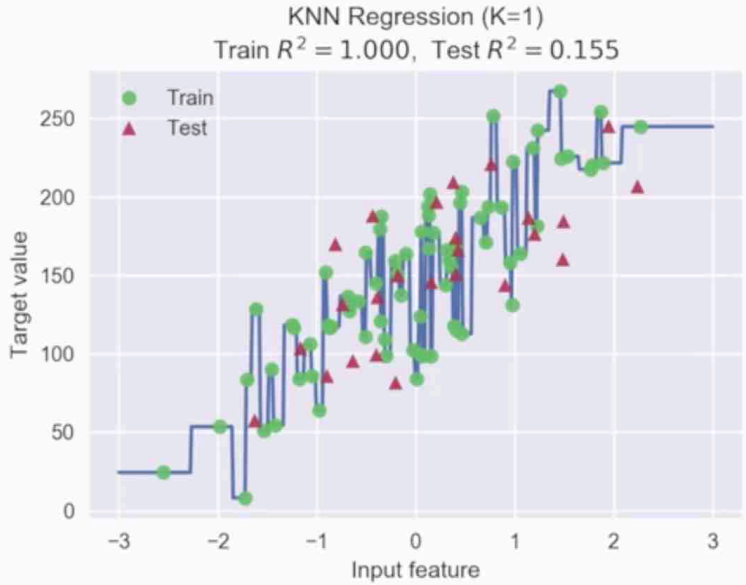
The R^2 ("r-squared") regression score

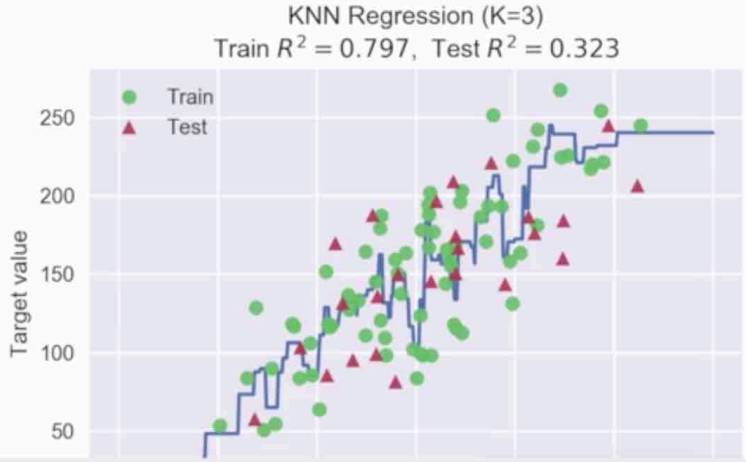
- Measures how well a prediction model for regression fits the given data.
- The score is between 0 and 1:
  - *A value of 0 corresponds to a constant model that predicts the mean value of all training target values.*
  - *A value of 1 corresponds to perfect prediction*
- Also known as "coefficient of determination"

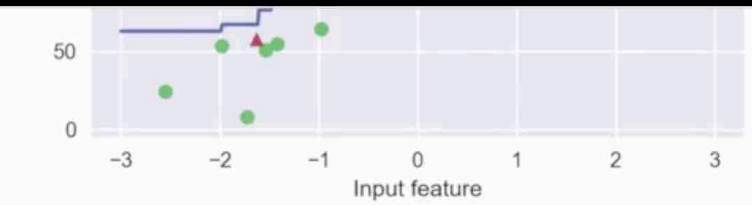
Regression model complexity as a function of K

In [ ]:

```
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)
```

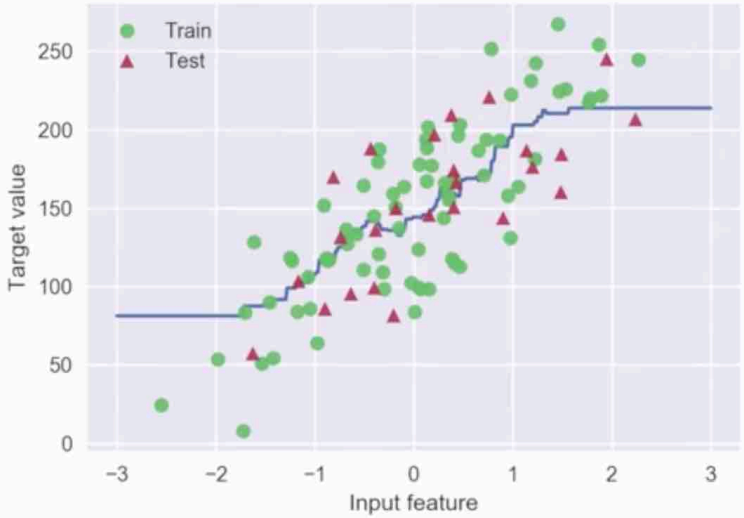

KNN Regression (K=1)
Train $R^2 = 1.000$, Test $R^2 = 0.155$
Train
Test
Target value
Input feature
0
50
100
150
200
250
−3
−2
−1
0
1
2
3


KNN Regression (K=3)
Train $R^2 = 0.797$, Test $R^2 = 0.323$
Train
Test
Target value
50
100
150
200
250



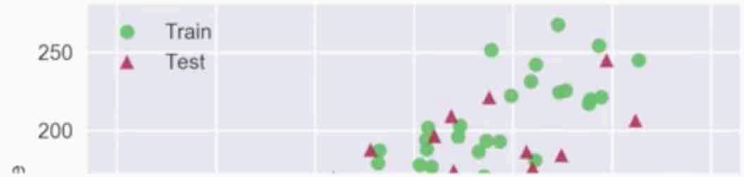
KNN Regression (K=15)

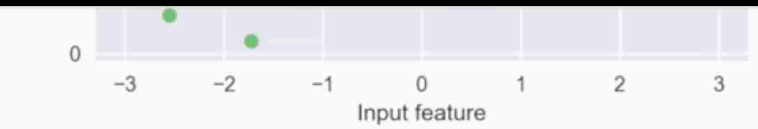
Train $R^2 = 0.647$, Test $R^2 = 0.485$

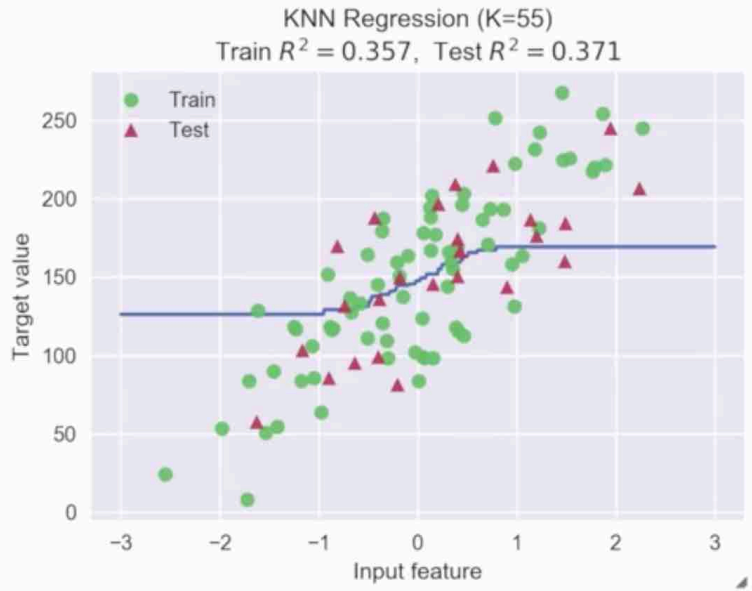


KNN Regression (K=55)

Train $R^2 = 0.357$, Test $R^2 = 0.371$




0
−3
−2
−1
0
1
2
3
Input feature


KNN Regression (K=55)
Train $R^2 = 0.357$, Test $R^2 = 0.371$
Train
Test
Target value
250
200
150
100
50
0
−3
−2
−1
0
1
2
3
Input feature

In [ ]:

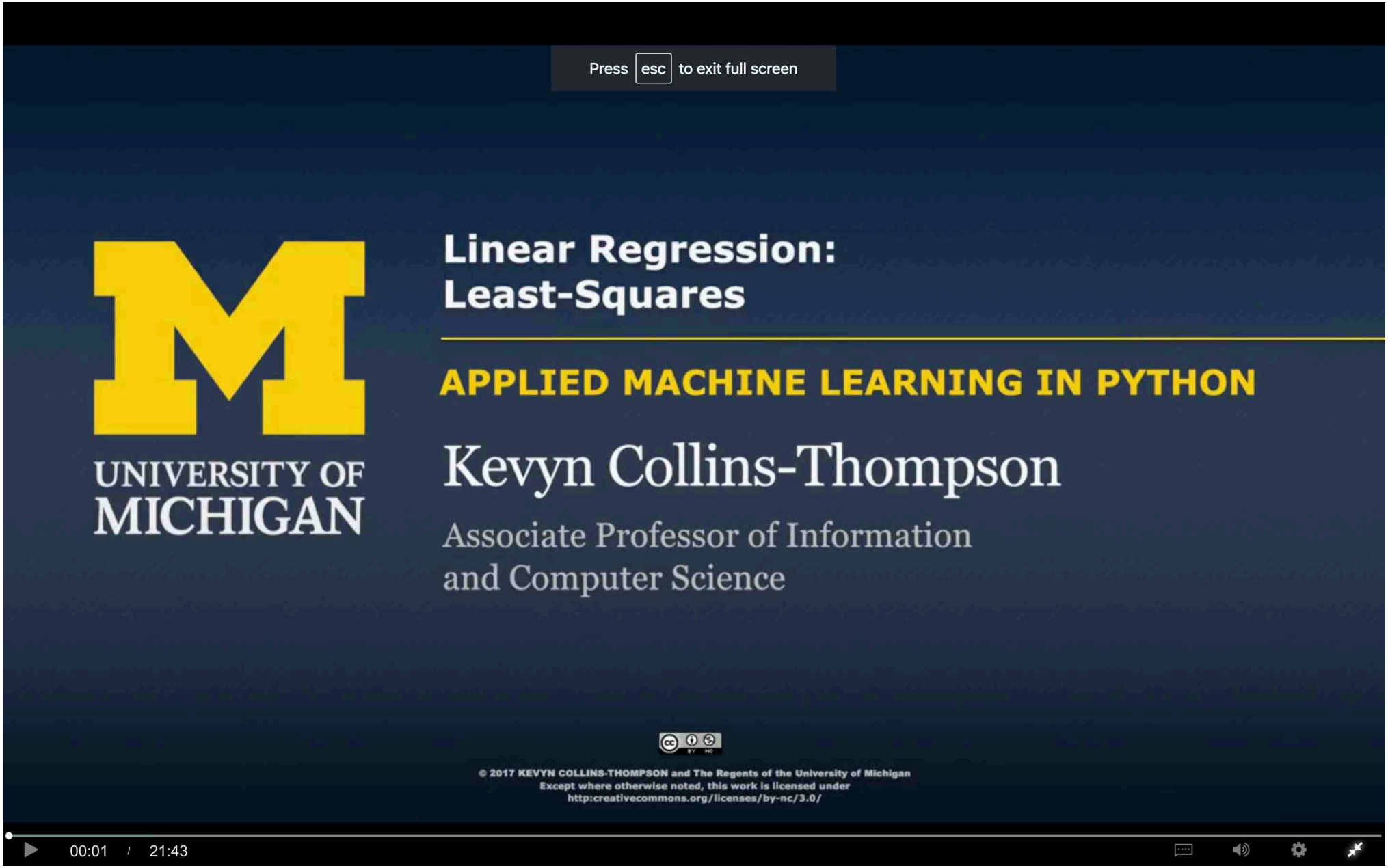
KNeighborsClassifier and KNeighborsRegressor: important parameters

Model complexity

- *n_neighbors* : number of nearest neighbors (k) to consider
  - *Default = 5*

Model fitting

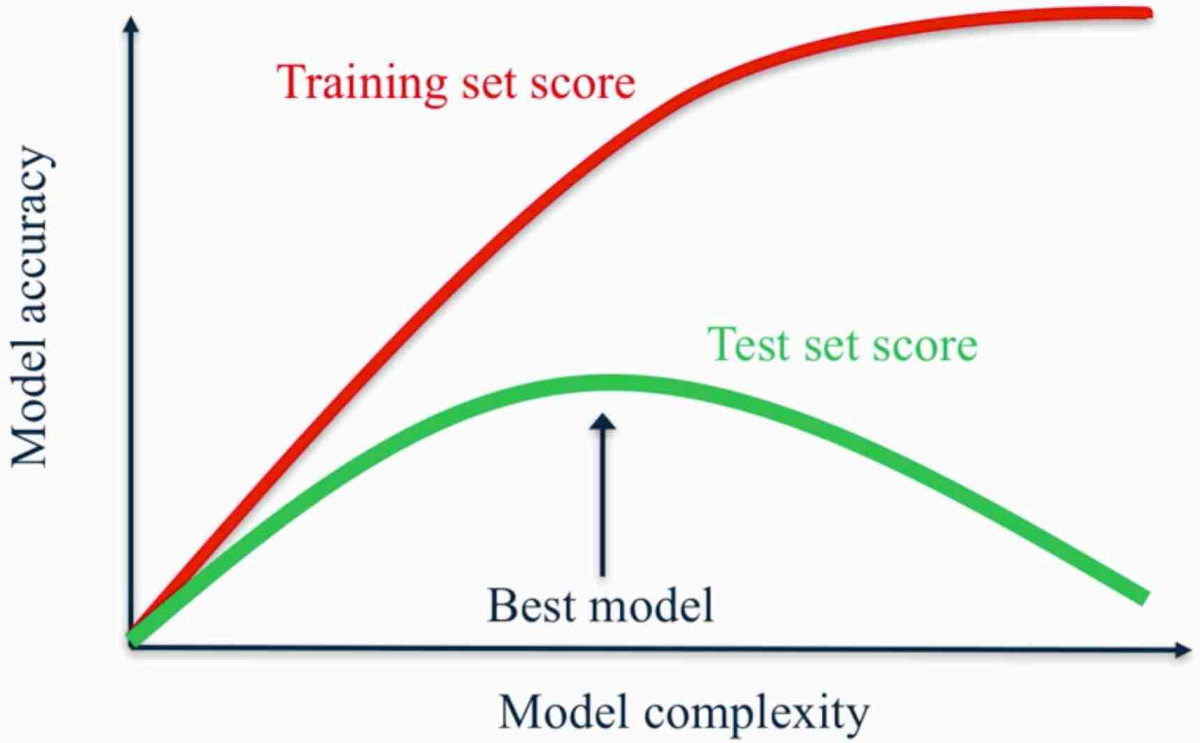
- *metric: distance function between data points*
  - *Default: Minkowski distance with power parameter p = 2 (Euclidean)*


Press esc to exit full screen
UNIVERSITY OF MICHIGAN
Linear Regression: Least-Squares
APPLIED MACHINE LEARNING IN PYTHON
Kevyn Collins-Thompson
Associate Professor of Information and Computer Science
© 2017 KEVYN COLLINS-THOMPSON and The Regents of the University of Michigan
Except where otherwise noted, this work is licensed under
http:creativecommons.org/licenses/by-nc/3.0/
00:01 / 21:43

Supervised learning methods: Overview

- To start with, we'll look at two simple but powerful prediction algorithms:
  - *K-nearest neighbors (review from week 1, plus regression)*
  - *Linear model fit using least-squares*
- These represent two complementary approaches to supervised learning:
  - *K-nearest neighbors makes few assumptions about the structure of the data and gives potentially accurate but sometimes unstable predictions (sensitive to small changes in the training data).*
  - *Linear models make strong assumptions about the structure of the data and give stable but potentially inaccurate predictions.*

The relationship between model complexity and training/test performance



Linear Models

- A linear model is a sum of weighted variables that predicts a target output value given an input data instance. Example: *predicting housing prices*
  - *House features: taxes per year (X_{TAX}), age in years (X_{AGE})*

$$\widehat{Y_{PRICE}} = 212000 + 109\, X_{TAX} - 2000\, X_{AGE}$$

  - *A house with feature values (X_{TAX}, X_{AGE}) of (10000, 75) would have a predicted selling price of:*

$$\widehat{Y_{PRICE}} = 212000 + 109 \cdot \mathbf{10000} - 2000 \cdot \mathbf{75} = \mathbf{1{,}152{,}000}$$

Linear Regression is an Example of a Linear Model

Input instance – feature vector: $\boldsymbol{x} = (x_0, x_1, \ldots, x_n)$

Predicted output: $\hat{y} = \widehat{w_0} x_0 + \widehat{w_1} x_1 + \cdots \widehat{w_n} x_n + \hat{b}$

Parameters to estimate:

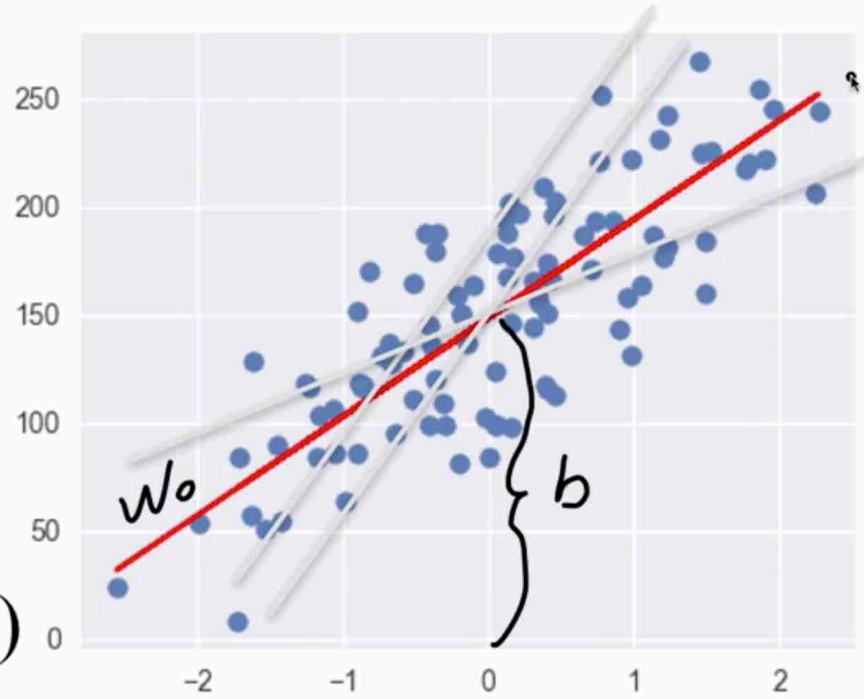
- $\hat{\boldsymbol{w}} = (\widehat{w_0}, \cdots, \widehat{w_n})$: feature weights/ model coefficients
- $\hat{\boldsymbol{b}}$: constant bias term / intercept

A linear regression model with one variable (feature)

Input instance: $\boldsymbol{x} = (x_0)$

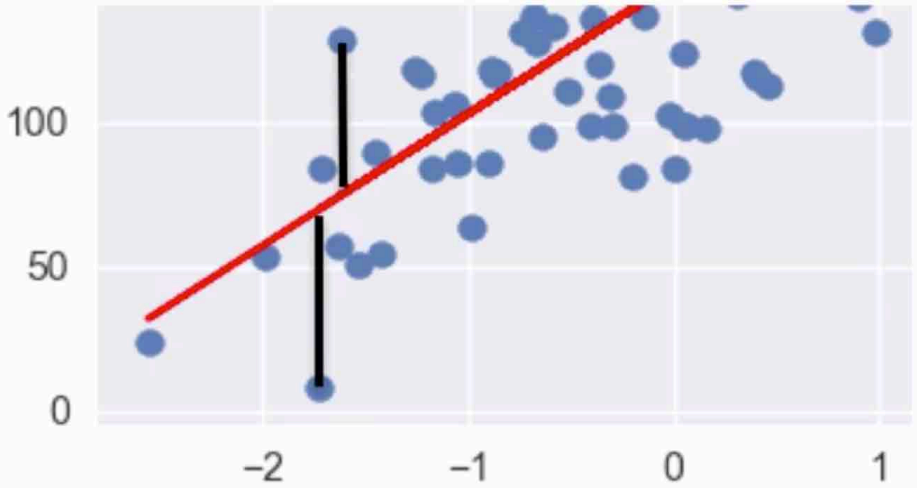
Predicted output: $\hat{y} = w_0 x_0 + b$

Parameters to estimate:
- w_0 (Slope)
- b (y-intercept)



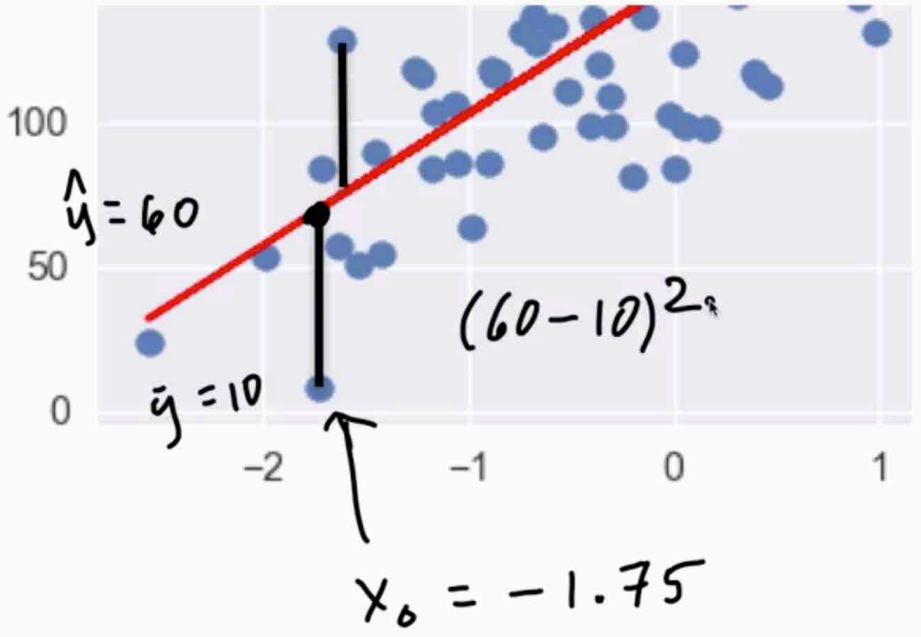
Least-squares Linear Regression ("Ordinary least-squares")

- Finds w and b that minimizes the mean squared error of the model: the sum of squared differences between predicted target and actual target values.
- No parameters to control model complexity.



Least-squares Linear Regression ("Ordinary least-squares")

- Finds w and b that minimizes the mean squared error of the model: the sum of squared differences between predicted target and actual target values.
- No parameters to control model complexity.

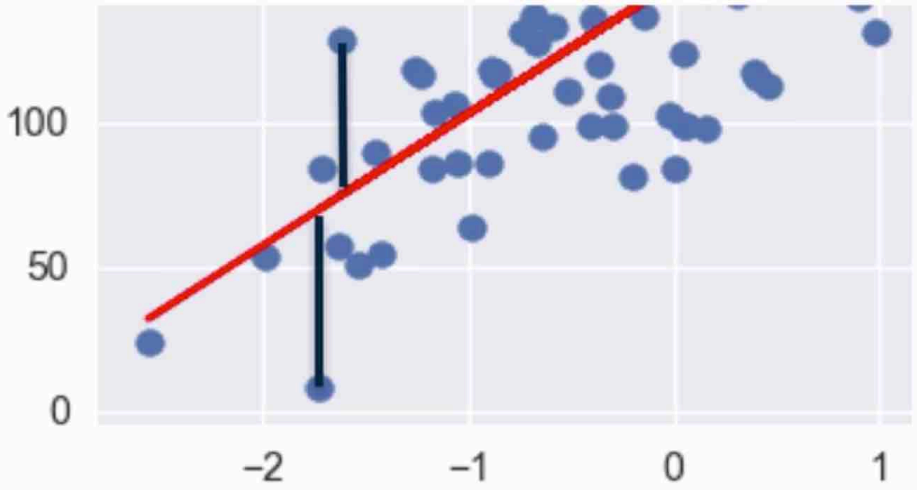


How are Linear Regression Parameters *w*, *b* Estimated?

- Parameters are estimated from training data.
- There are many different ways to estimate *w* and *b*:
  - *Different methods correspond to different "fit" criteria and goals and ways to control model complexity.*
- The learning algorithm finds the parameters that optimize an objective function, typically to minimize some kind of loss function of the predicted target values *vs.* actual target values.

Least-Squares Linear Regression ("Ordinary Least-Squares")

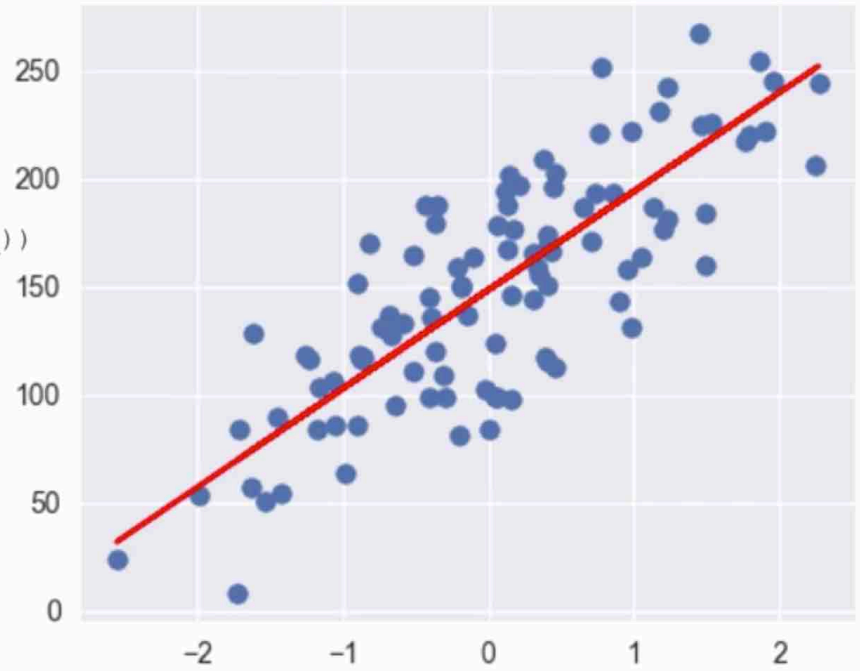
- Finds *w* and *b* that minimizes the sum of squared differences (RSS) over the training data between predicted target and actual target values.
- a.k.a. mean squared error of the linear model
- No parameters to control model complexity.



$$RSS(\boldsymbol{w}, b) = \sum_{\{i=1\}}^{N} (\boldsymbol{y}_i - (\boldsymbol{w} \cdot \boldsymbol{x}_i + b))^2$$

Least-Squares Linear Regression in Scikit-Learn

```
from sklearn.linear_model import LinearRegression

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

linreg = LinearRegression().fit(X_train, y_train)

print("linear model intercept (b): {}".format(linreg.intercept_))
print("linear model coeff (w): {}".format(linreg.coef_))
```



linreg.coef_ linreg.intercept_

$$\hat{y} = \boldsymbol{w_0} x_0 + b$$

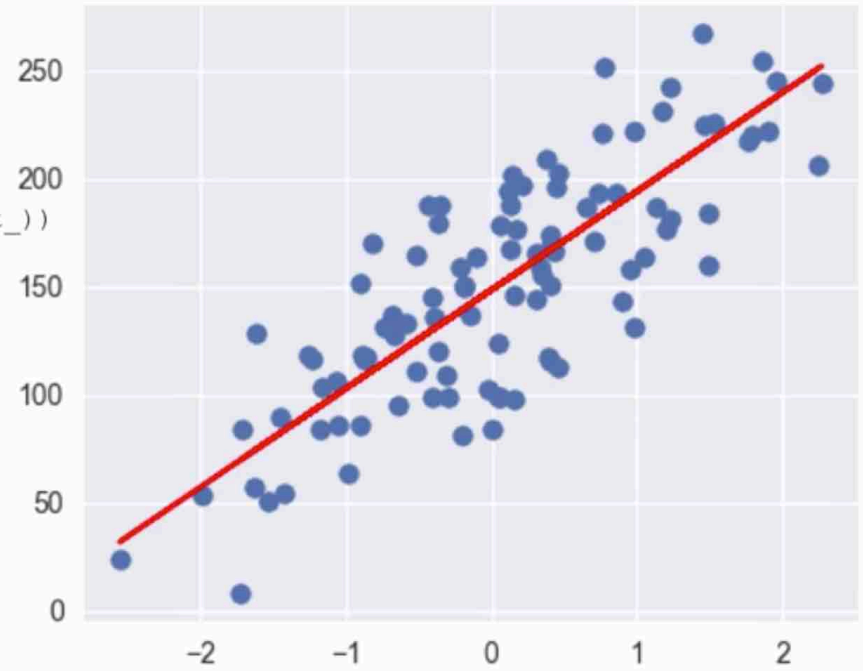
Least-Squares Linear Regression in Scikit-Learn

```
from sklearn.linear_model import LinearRegression

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

linreg = LinearRegression().fit(X_train, y_train)

print("linear model intercept (b): {}".format(linreg.intercept_))
print("linear model coeff (w): {}".format(linreg.coef_))
```

Underscore denotes a quantity derived from training data, as opposed to a user setting.

`linreg.coef_` `linreg.intercept_`

$$\hat{y} = \boldsymbol{w_0} x_0 + b$$



Linear regression

In [7]:
```
from sklearn.linear_model import LinearRegression

X_train, X_test, y_train, y_test = train_test_split(X_R1, y_R1,
                                                    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)))
```

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

In [ ]:

Linear regression: example plot

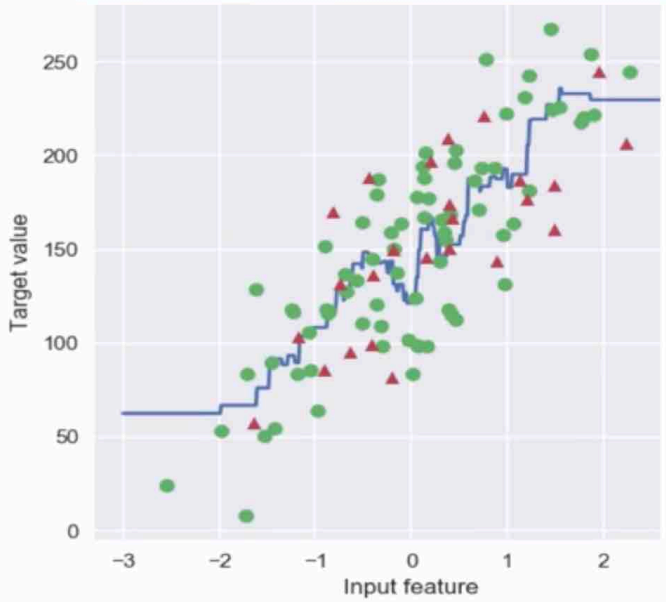
In [8]:
```
plt.figure(figsize=(5,4))
plt.scatter(X_R1, y_R1, marker= 'o', s=50, alpha=0.8)
plt.plot(X_R1, linreg.coef_ * X_R1 + linreg.intercept_, 'r-')
plt.title('Least-squares linear regression')
plt.xlabel('Feature value (x)')
plt.ylabel('Target value (y)')
plt.show()
```

Figure 10

K-NN Regression vs Least-Squares Linear Regression

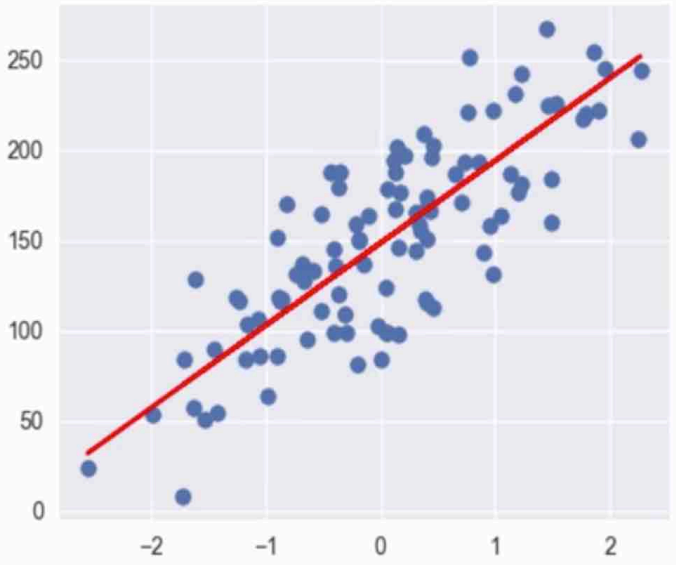
k-NN (k=7)

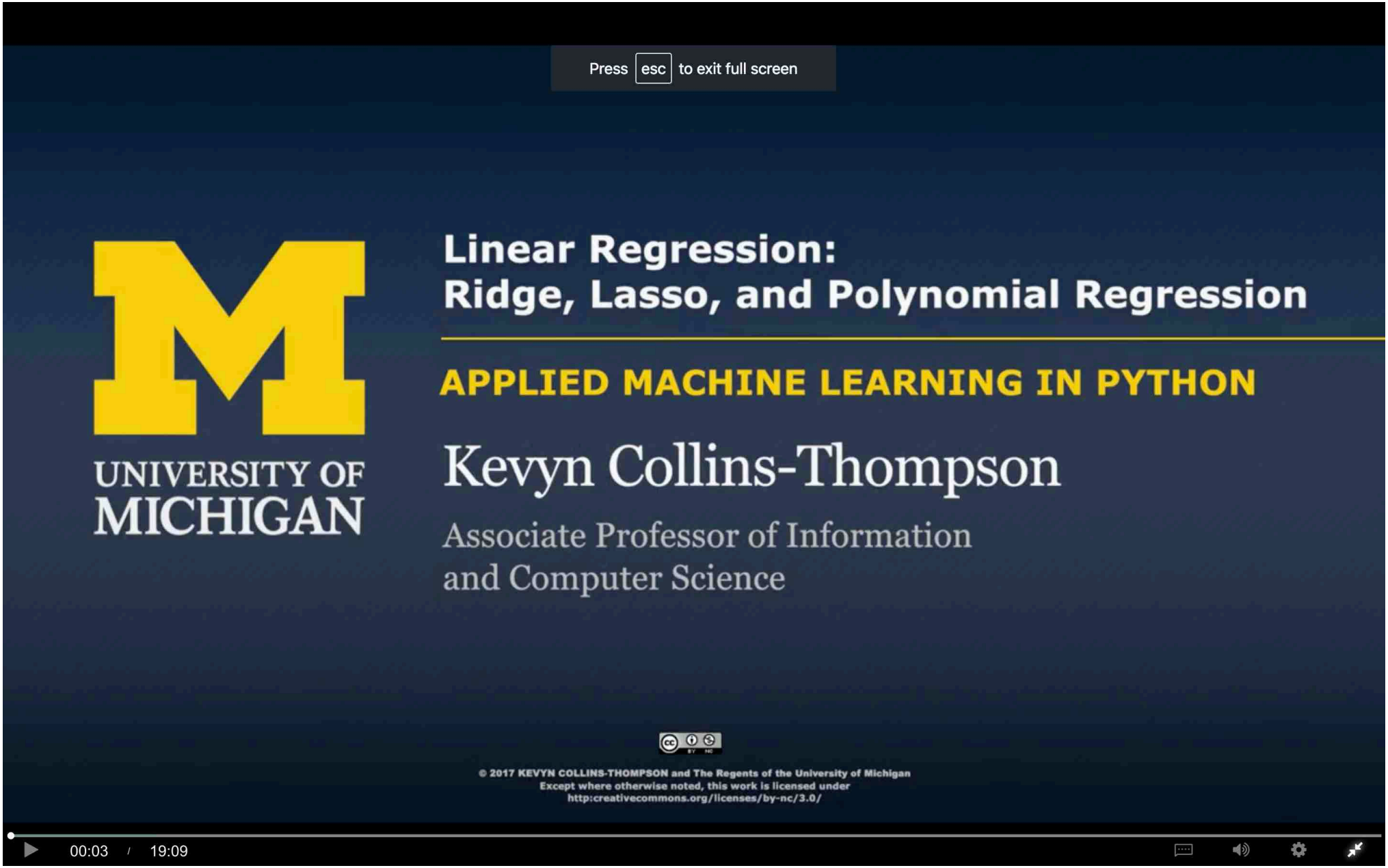
```
R-squared score (training): 0.720
R-squared score (test): 0.471
```



LS linear

```
R-squared score (training): 0.679
R-squared score (test): 0.492
```




Press esc to exit full screen
UNIVERSITY OF MICHIGAN
Linear Regression:
Ridge, Lasso, and Polynomial Regression
APPLIED MACHINE LEARNING IN PYTHON
Kevyn Collins-Thompson
Associate Professor of Information and Computer Science
© 2017 KEVYN COLLINS-THOMPSON and The Regents of the University of Michigan
Except where otherwise noted, this work is licensed under
http:creativecommons.org/licenses/by-nc/3.0/
00:03 / 19:09

Ridge Regression

- Ridge regression learns *w*, *b* using the same least-squares criterion but adds a penalty for large variations in *w* parameters

$$RSS_{RIDGE}(\boldsymbol{w}, b) = \sum_{\{i=1\}}^{N} (\boldsymbol{y}_i - (\boldsymbol{w} \cdot \boldsymbol{x}_i + b))^2 + \alpha \sum_{\{j=1\}}^{p} w_j^2$$

- Once the parameters are learned, the ridge regression prediction formula is the same as ordinary least-squares.
- The addition of a parameter penalty is called regularization. Regularization prevents overfitting by restricting the model, typically to reduce its complexity.
- Ridge regression uses L2 regularization: minimize sum of squares of *w* entries
- The influence of the regularization term is controlled by the α parameter.
- Higher alpha means more regularization and simpler models.

Ridge regression

In [10]:
```
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)))
```

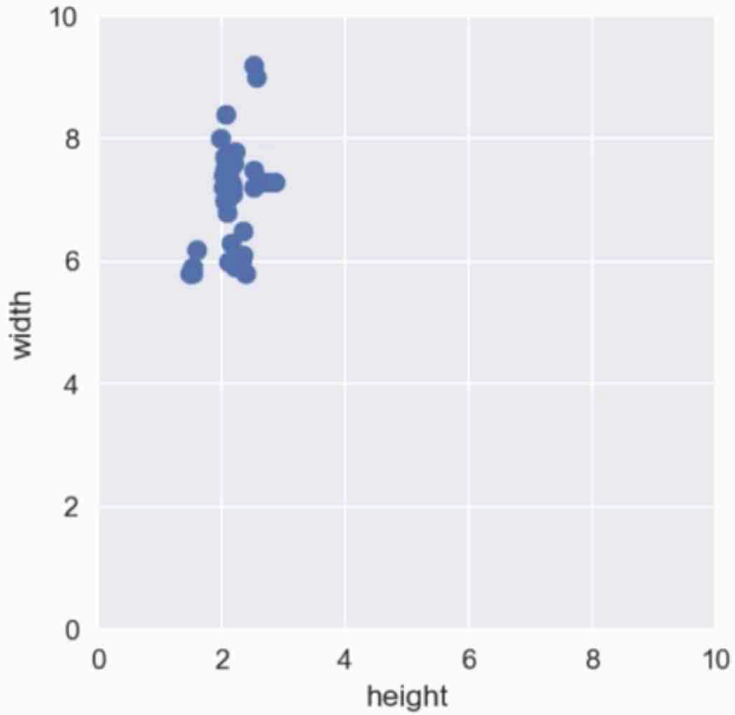
```
Crime dataset
ridge regression linear model intercept: -3352.4230358466525
ridge regression linear model coeff:
[  1.95e-03   2.19e+01   9.56e+00  -3.59e+01   6.36e+00  -1.97e+01
  -2.81e-03   1.66e+00  -6.61e-03  -6.95e+00   1.72e+01  -5.63e+00
   8.84e+00   6.79e-01  -7.34e+00   6.70e-03   9.79e-04   5.01e-03
```

The Need for Feature Normalization

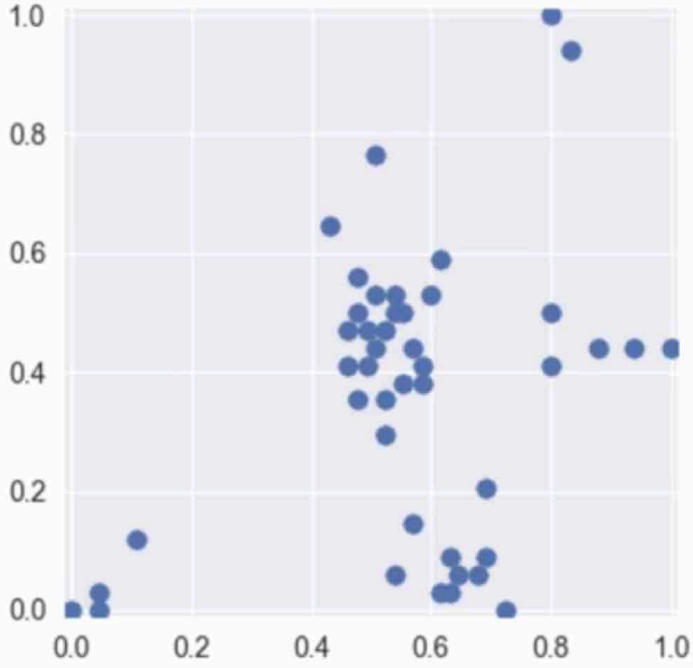
- Important for some machine learning methods that all features are on the same scale (e.g. faster convergence in learning, more uniform or 'fair' influence for all weights)
  - *e.g. regularized regression, k-NN, support vector machines, neural networks, …*
- Can also depend on the data. More on feature engineering later in the course. For now, we do MinMax scaling of the features:
  - *For each feature x_i : compute the min value x_i^{MIN} and the max value x_i^{MAX} achieved across all instances in the training set.*
  - *For each feature: transform a given feature x_i value to a scaled version x_i' using the formula*

$$x_i' = (x_i - x_i^{MIN})/(x_i^{MAX} - x_i^{MIN})$$

Feature Normalization with MinMaxScaler



Unnormalized data points



Normalized with MinMaxScaler

Using a scaler object: fit and transform methods

```
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler()
scaler.fit(X_train)
X_train_scaled = scaler.transform(X_train)
X_test_scaled = scaler.transform(X_test)
clf = Ridge().fit(X_train_scaled, y_train)
r2_score = clf.score(X_test_scaled, y_test)
```

Tip: It can be more efficient to do fitting and transforming together on the training set using the `fit_transform` method.

```
scaler = MinMaxScaler()
X_train_scaled = scaler.fit_transform(X_train)
```

Feature Normalization: The test set must use identical scaling to the training set

- Fit the scaler using the training set, then apply the same scaler to transform the test set.
- Do not scale the training and test sets using different scalers: this could lead to random skew in the data.
- Do not fit the scaler using any part of the test data: referencing the test data can lead to a form of *data leakage*. More on this issue later in the course.

Ridge regression with feature normalization

In [ ]:

```
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler()

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)
X_test_scaled = scaler.transform(X_test)

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)))
```

```
print('Number of non-zero features: {}'
      .format(np.sum(linridge.coef_ != 0)))
```

```
Crime dataset
ridge regression linear model intercept: 933.3906385044163
ridge regression linear model coeff:
[  88.69   16.49  -50.3   -82.91  -65.9    -2.28   87.74  150.95   18.88
  -31.06  -43.14 -189.44   -4.53  107.98  -76.53    2.86   34.95   90.14
   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  -284.5
 -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   53.81
   49.02   -7.62   55.14  -52.09  123.39   77.13   45.5   184.91  -91.36
    1.08  234.09   10.39   94.72  167.92  -25.14   -1.18   14.6    36.77
   53.2   -78.86   -5.9    26.05  115.15   68.74   68.29   16.53  -97.91
  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
```

In [ ]:

Ridge regression with regularization parameter: alpha

In [12]:
```
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}\n\
num 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

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
```

```
Ridge regression: effect of alpha regularization parameter

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 is another form of regularized linear regression that uses an L1 regularization penalty for training (instead of ridge's L2 penalty)

- L1 penalty: Minimize the sum of the absolute values of the coefficients

$$RSS_{LASSO}(\boldsymbol{w}, b) = \sum_{\{i=1\}}^{N} (\boldsymbol{y}_i - (\boldsymbol{w} \cdot \boldsymbol{x}_i + b))^2 + \alpha \sum_{\{j=1\}}^{p} |w_j|$$

- This has the effect of setting parameter weights in *w* to zero for the least influential variables. This is called a sparse solution: a kind of feature selection
- The parameter α controls amount of L1 regularization (default = 1.0).
- The prediction formula is the same as ordinary least-squares.
- When to use ridge vs lasso regression:
  - *Many small/medium sized effects: use ridge.*
  - *Only a few variables with medium/large effect: use lasso.*

Lasso regression

In [ ]:

```
from sklearn.linear_model import Lasso
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler()

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)
X_test_scaled = scaler.transform(X_test)

linlasso = Lasso(alpha=2.0, max_iter = 10000).fit(X_train_scaled, y_train)

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_tra)))
```

In [ ]:

```
from sklearn.linear_model import Lasso
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler()

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)
X_test_scaled = scaler.transform(X_test)

linlasso = Lasso(alpha=2.0, max_iter = 10000).fit(X_train_scaled, y_train)

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: 1186.612061998579
lasso regression linear model coeff:
[    0.       0.      -0.    -168.18    -0.      -0.       0.     119.69
     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.      73.14
     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
```

Lasso Regression on the Communities and Crime Dataset

For alpha = 2.0, 20 out of 88 features have non-zero weight.

Top features (sorted by abs. magnitude):

```
PctKidsBornNeverMar, 1488.365 # percentage of kids born to people who never married
PctKids2Par, -1188.740 # percentage of kids in family housing with two parents
HousVacant, 459.538  # number of vacant households
PctPersDenseHous, 339.045 # percent of persons in dense housing (more than 1 person/room)
NumInShelters, 264.932 # number of people in homeless shelters
```

Lasso regression with regularization parameter: alpha

In [14]:
```
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}\n\
Features 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
```

```
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
```

```
In [ ]:
```

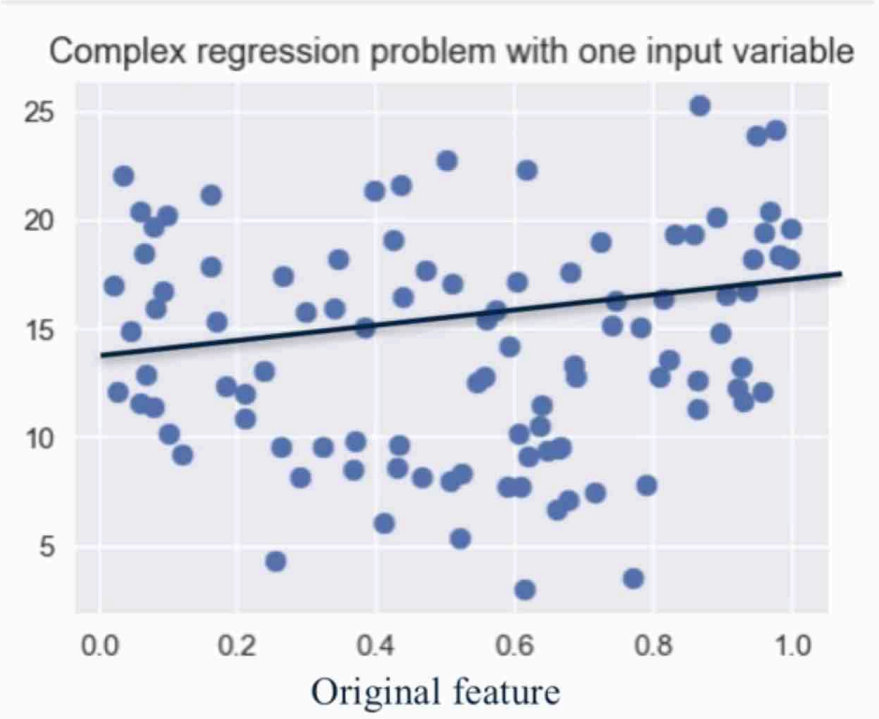
Polynomial Features with Linear Regression

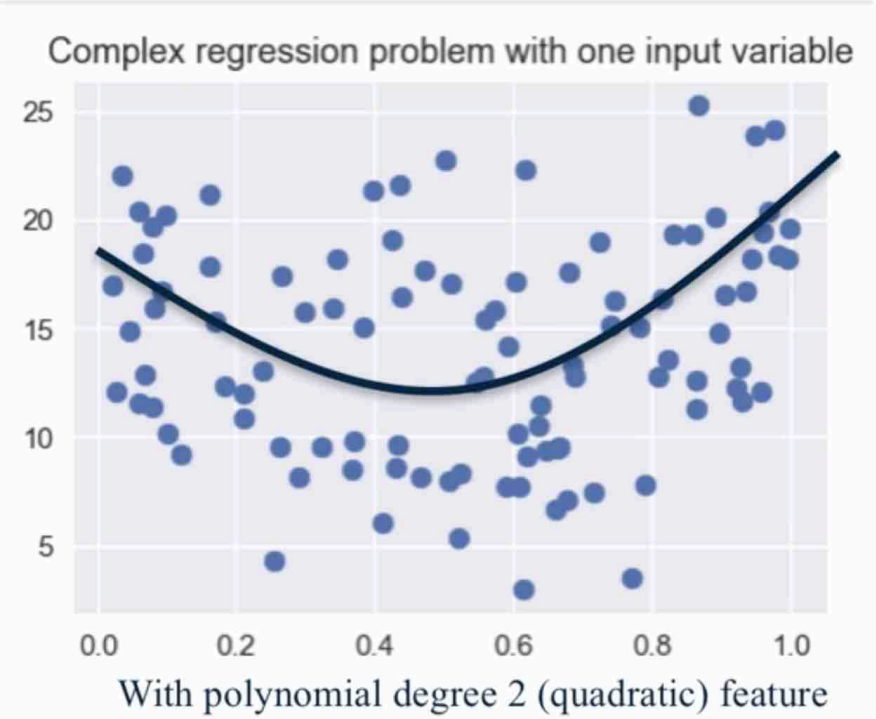
$$\boldsymbol{x} = (x_0, x_1) \Rightarrow \boldsymbol{x}' = (x_0, x_1, x_0^2, x_0 x_1, x_1^2)$$

$$\hat{y} = \hat{w}_0 x_0 + \hat{w}_1 x_1 + \hat{w}_{00} x_0^2 + \hat{w}_{01} x_0 x_1 + \hat{w}_{11} x_1^2 + b$$

- Generate new features consisting of all polynomial combinations of the original two features (x_0, x_1).
- The *degree* of the polynomial specifies how many variables participate at a time in each new feature (above example: degree 2)
- This is still a weighted linear combination of features, so it's still a linear model, and can use same least-squares estimation method for *w* and *b*.

Least-Squares Polynomial Regression





Polynomial Features with Linear Regression

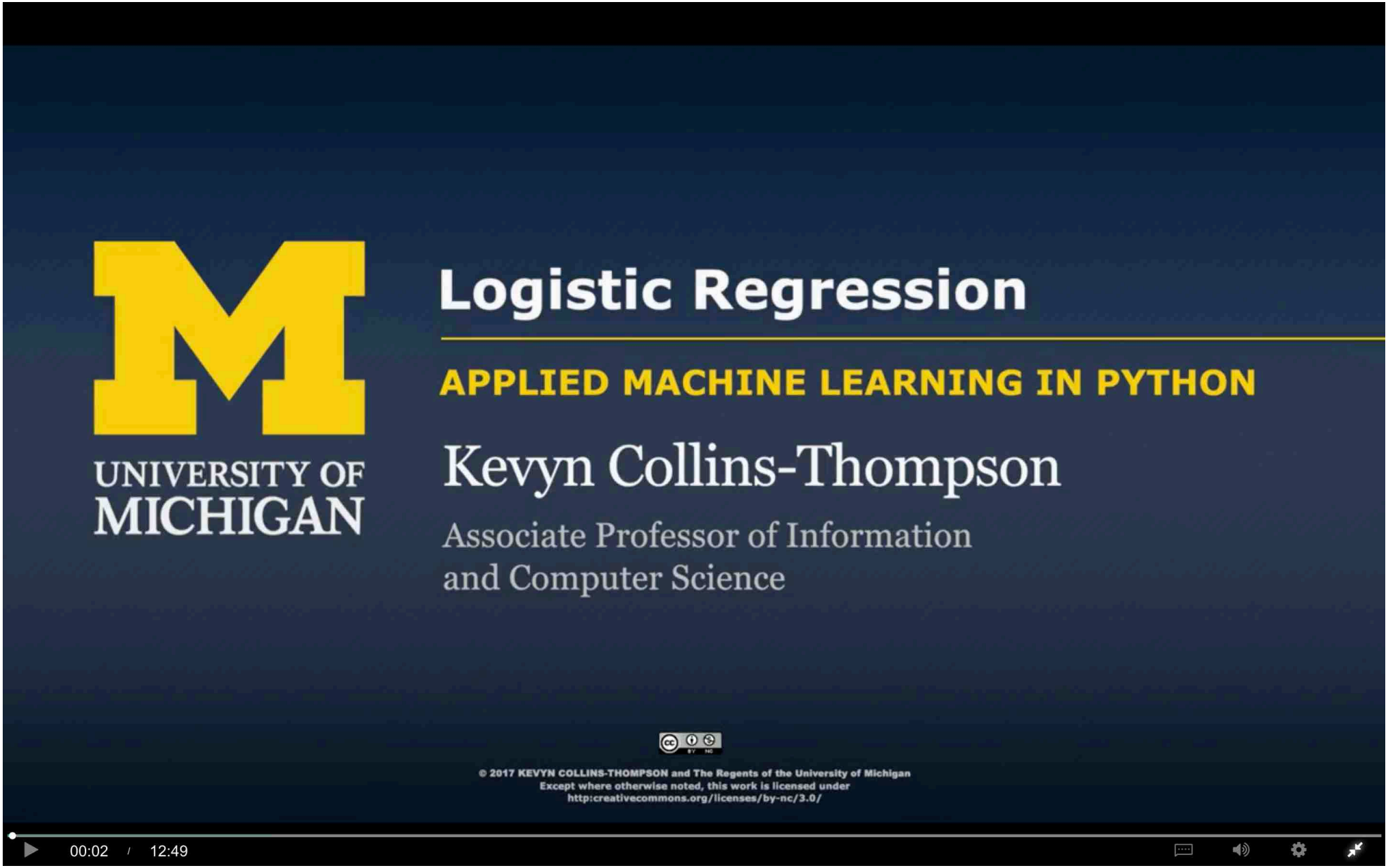
- **Why would we want to transform our data this way?**
  - *To capture interactions between the original features by adding them as features to the linear model.*
  - *To make a classification problem easier (we'll see this later).*
- **More generally, we can apply other non-linear transformations to create new features**
  - *(Technically, these are called non-linear basis functions)*
- **Beware of polynomial feature expansion with high as this can lead to complex models that overfit**
  - *Thus, polynomial feature expansion is often combined with a regularized learning method like ridge regression.*

Polynomial regression

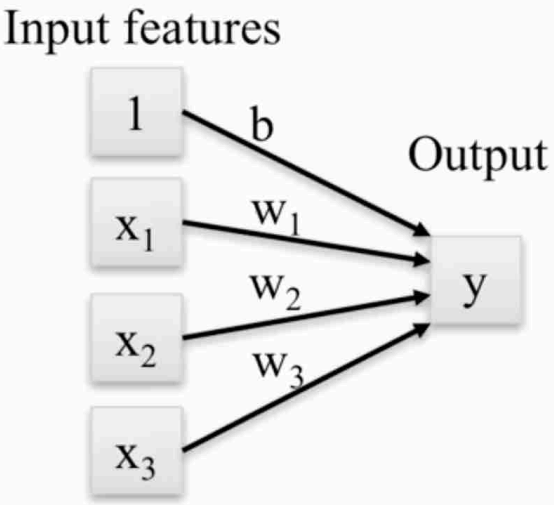
In [ ]:
```
from sklearn.linear_model import LinearRegression
from sklearn.linear_model import Ridge
from sklearn.preprocessing import PolynomialFeatures


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)))

print('\nNow we transform the original input data to add\n\
polynomial features up to degree 2 (quadratic)\n')
poly = PolynomialFeatures(degree=2)
```

```
                                                    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)))

print('\nNow we transform the original input data to add\n\
polynomial features up to degree 2 (quadratic)\n')
poly = PolynomialFeatures(degree=2)
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):')
```

```
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
```


UNIVERSITY OF MICHIGAN
Logistic Regression
APPLIED MACHINE LEARNING IN PYTHON
Kevyn Collins-Thompson
Associate Professor of Information and Computer Science
© 2017 KEVYN COLLINS-THOMPSON and The Regents of the University of Michigan
Except where otherwise noted, this work is licensed under
http:creativecommons.org/licenses/by-nc/3.0/
00:02 / 12:49

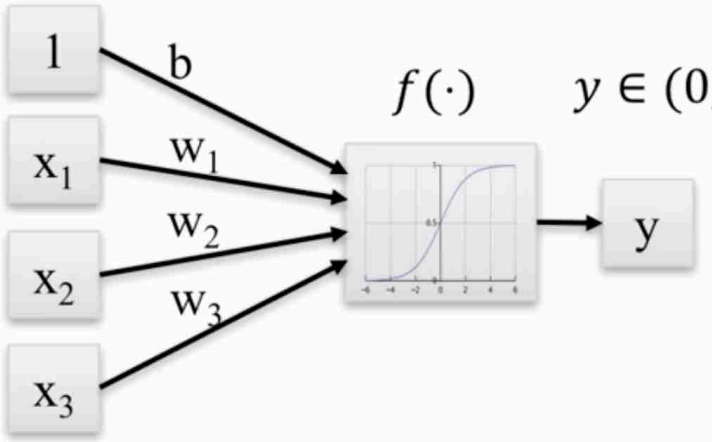
Linear Regression

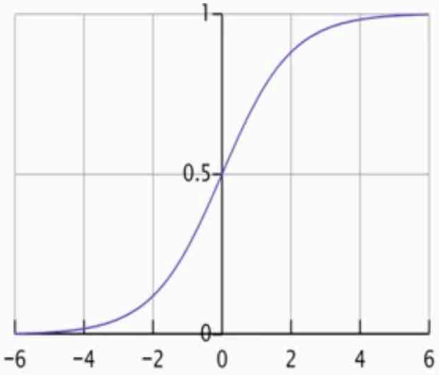


$$\hat{y} = \hat{b} + \hat{w}_1 \cdot x_1 + \cdots \hat{w}_n \cdot x_n$$

Linear models for classification: Logistic Regression

Input features



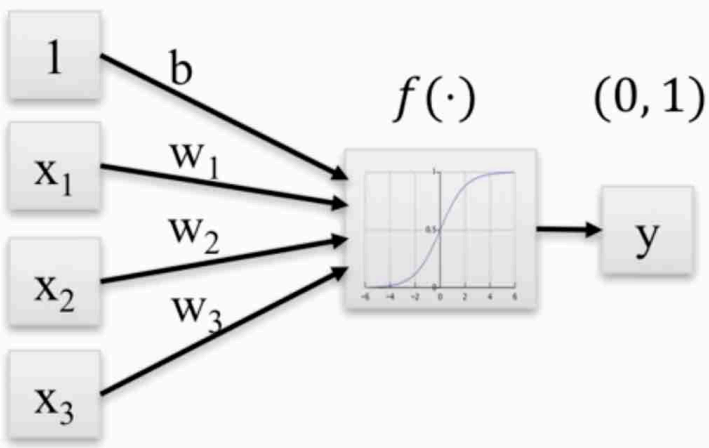


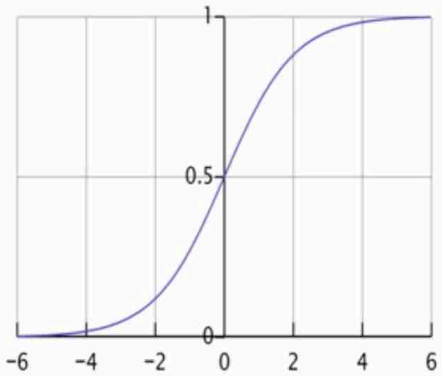
$$\hat{y} = \text{logistic}(\hat{b} + \hat{w}_1 \cdot x_1 + \cdots \hat{w}_n \cdot x_n)$$

$$= \frac{1}{1 + \exp\left[-(\hat{b} + \hat{w}_1 \cdot x_1 + \cdots \hat{w}_n \cdot x_n)\right]}$$

Linear models for classification: Logistic Regression

Input features

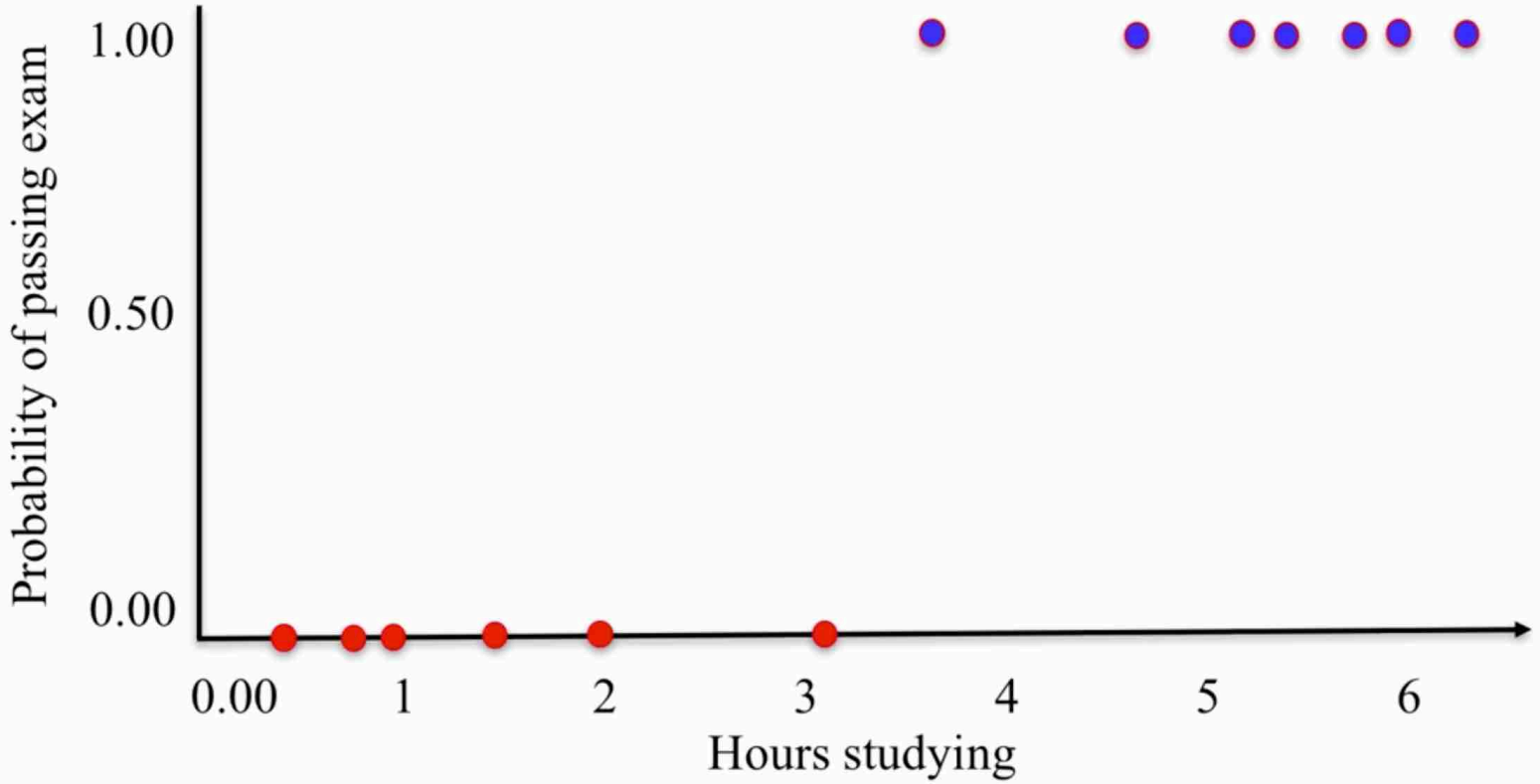




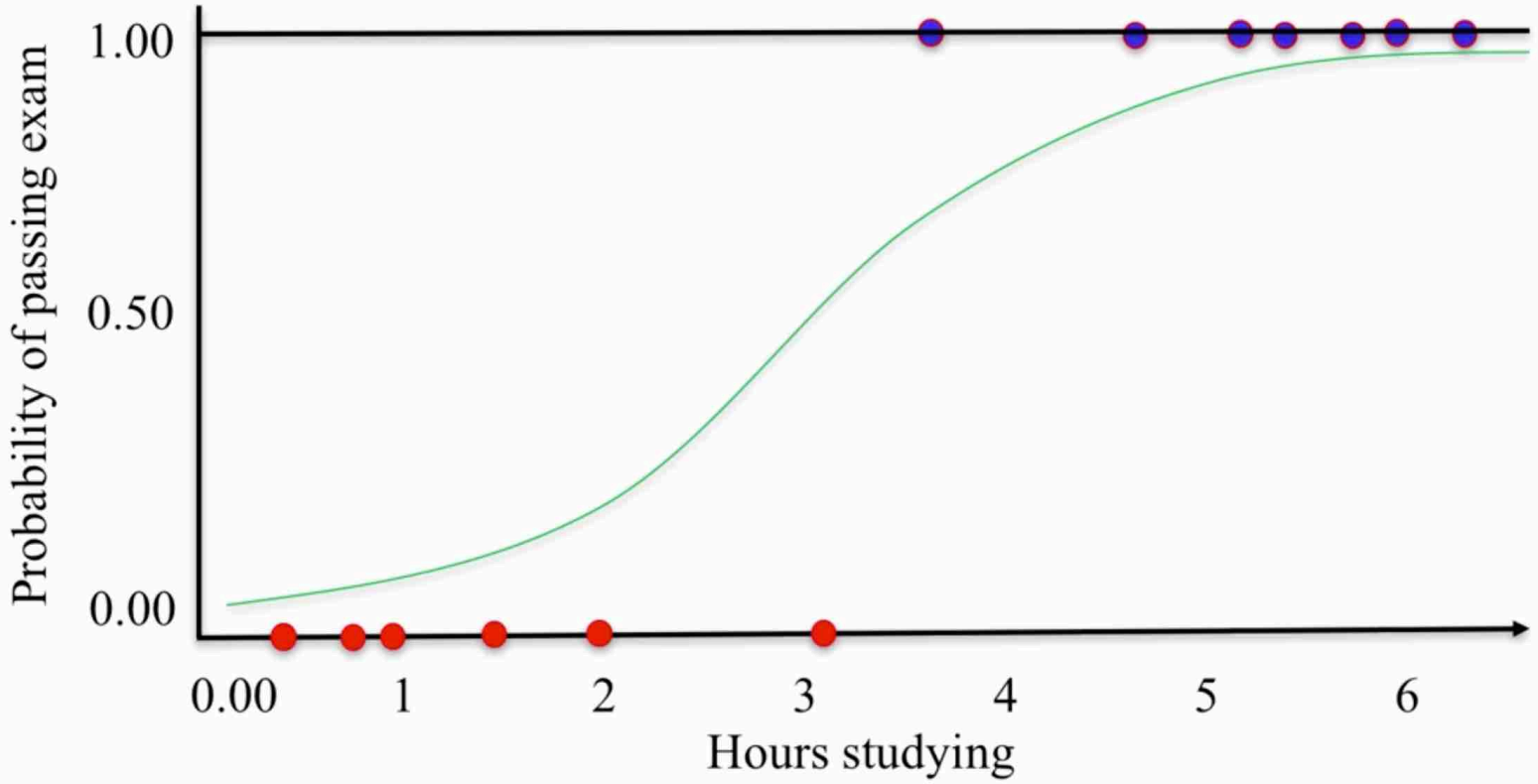
The logistic function transforms real-valued input to an output number y between 0 and 1, interpreted as the probability the input object belongs to the positive class, given its input features $(x_0, x_1, \ldots, x_n)$

$$\hat{y} = \text{logistic}(\hat{b} + \hat{w}_1 \cdot x_1 + \cdots \hat{w}_n \cdot x_n)$$
$$= \frac{1}{1 + \exp\left[-(\hat{b} + \hat{w}_1 \cdot x_1 + \cdots \hat{w}_n \cdot x_n)\right]}$$

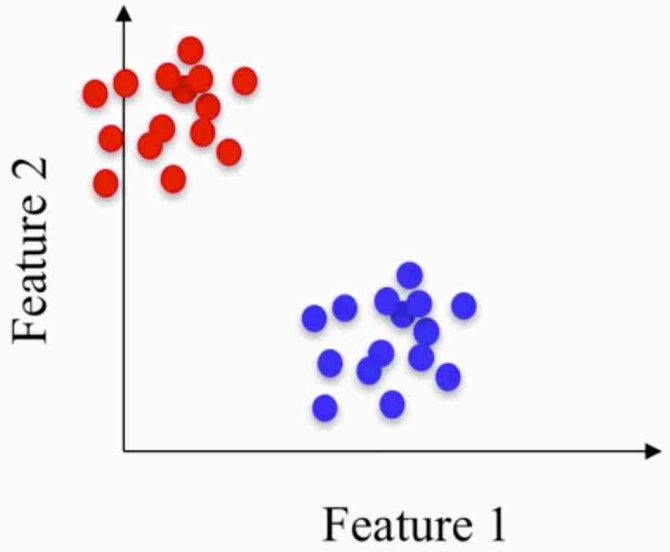
Linear models for classification: Logistic Regression



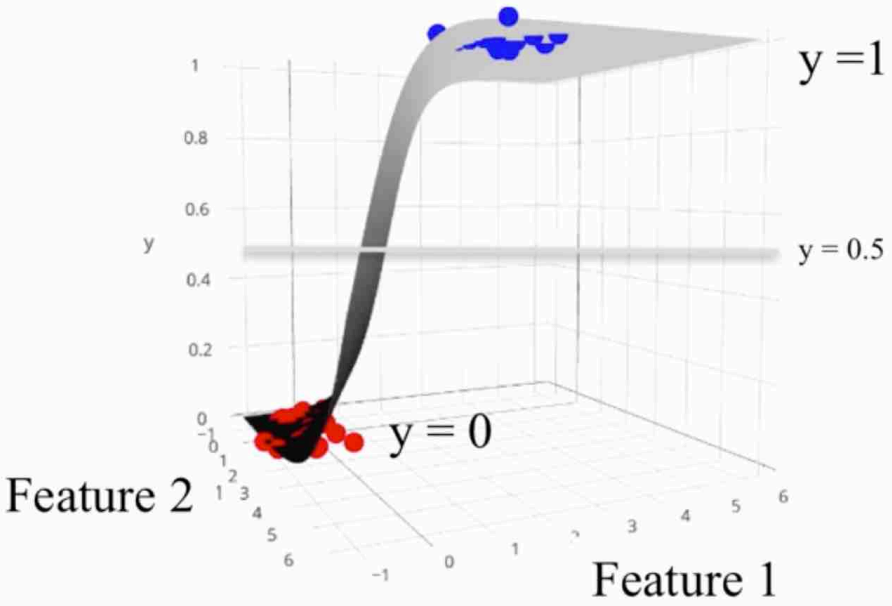
Linear models for classification: Logistic Regression



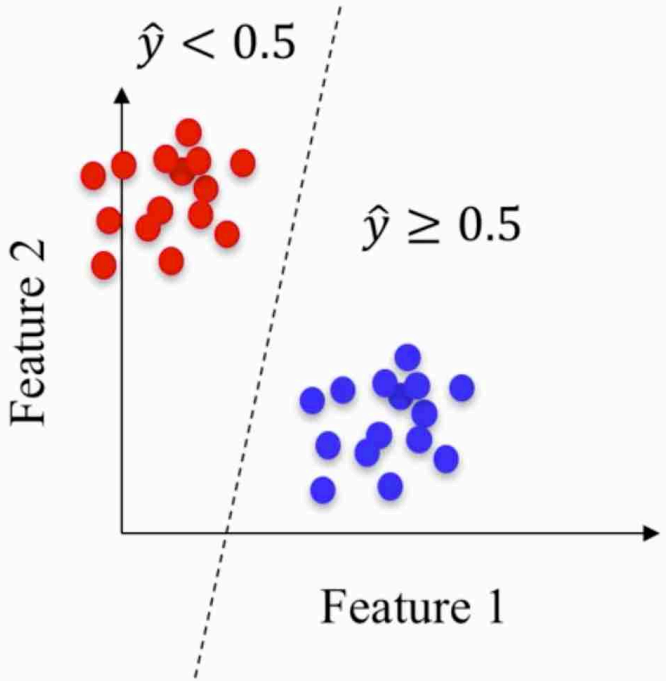
Logistic Regression for binary classification



Logistic Regression for binary classification



Logistic Regression for binary classification



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

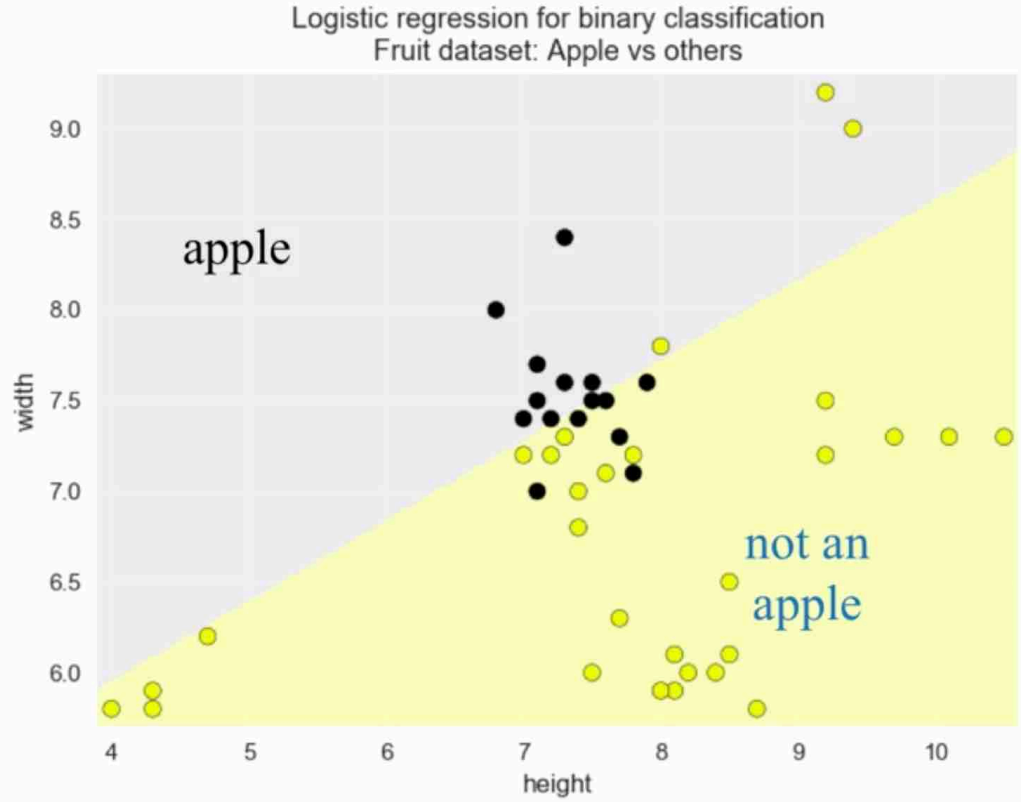
In [ ]:

```
from sklearn.linear_model import LogisticRegression
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
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)
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
```

```
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)
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]]))

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)))
```

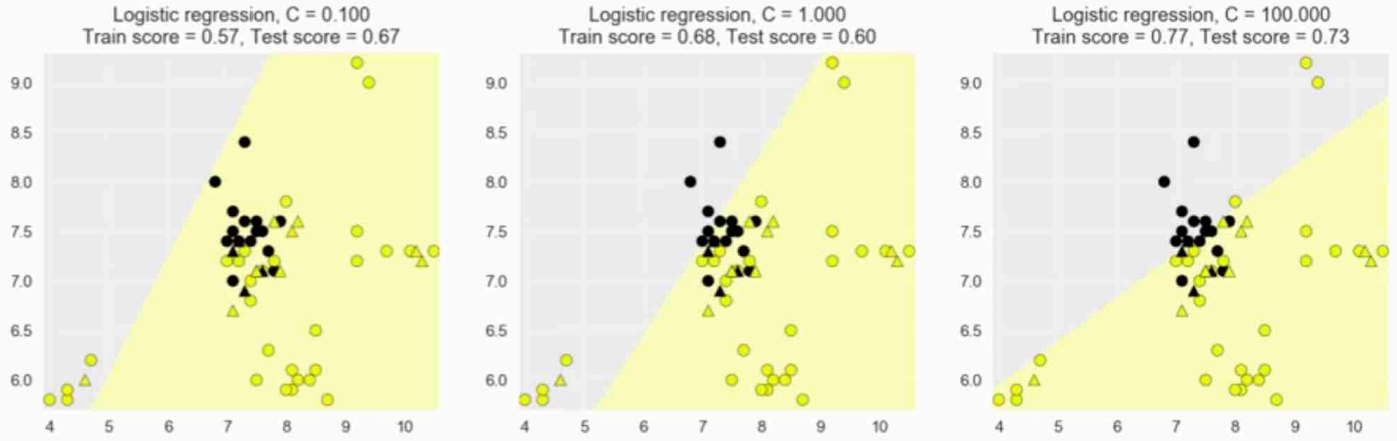
09:32 / 12:49

Simple logistic regression problem: two-class, two-feature version of the fruit dataset



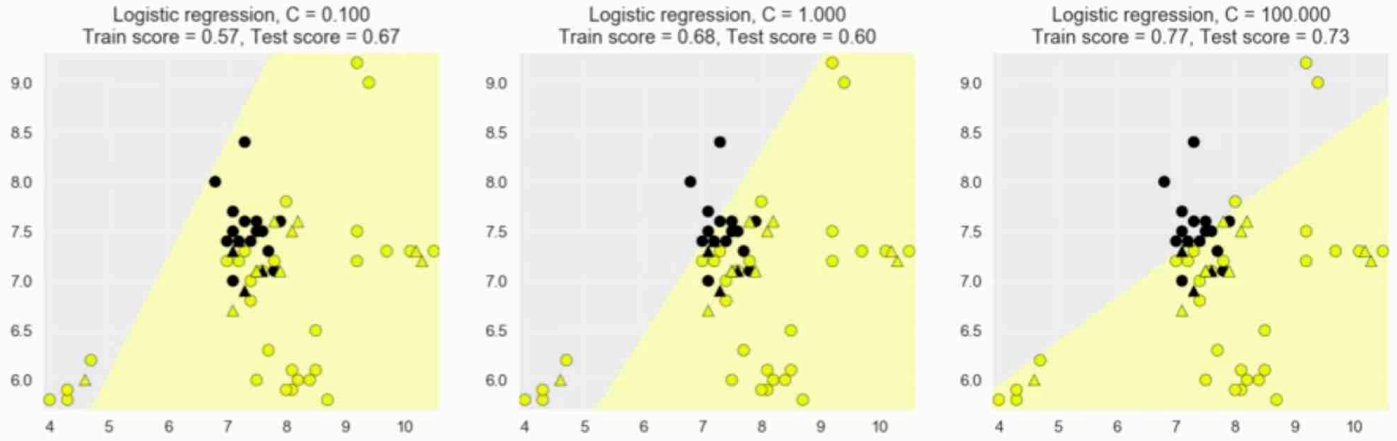
Logistic Regression: Regularization

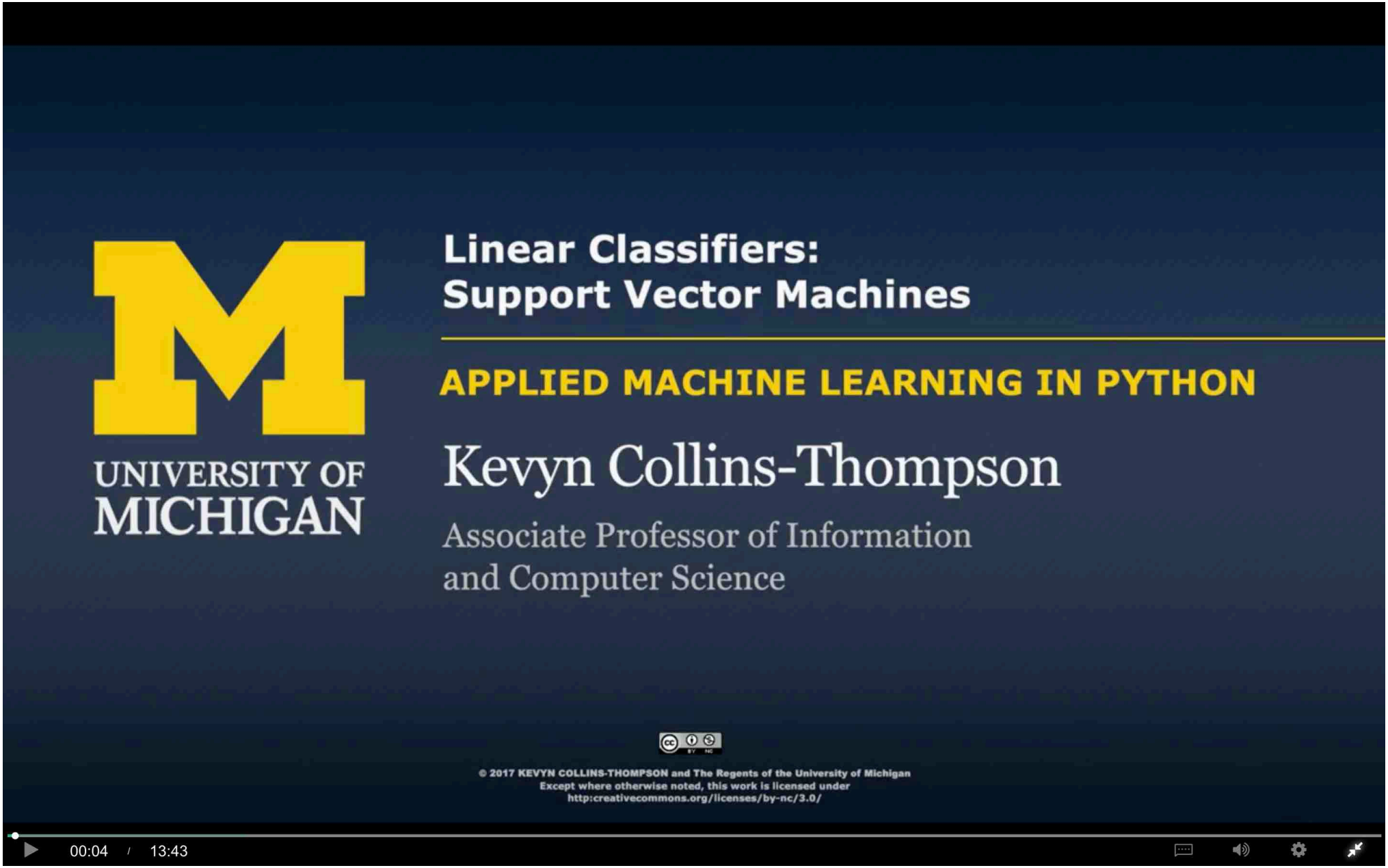
- L2 regularization is 'on' by default (like ridge regression)
- Parameter C controls amount of regularization (default 1.0)
- As with regularized linear regression, it can be important to normalize all features so that they are on the same scale.



Logistic Regression: Regularization

- L2 regularization is 'on' by default (like ridge regression)
- Parameter C controls amount of regularization (default 1.0)
- As with regularized linear regression, it can be important to normalize all features so that they are on the same scale.




UNIVERSITY OF MICHIGAN
Linear Classifiers:
Support Vector Machines
APPLIED MACHINE LEARNING IN PYTHON
Kevyn Collins-Thompson
Associate Professor of Information and Computer Science
© 2017 KEVYN COLLINS-THOMPSON and The Regents of the University of Michigan
Except where otherwise noted, this work is licensed under
http:creativecommons.org/licenses/by-nc/3.0/
00:04 / 13:43

Linear classifiers: how would you separate these two groups of training examples with a straight line?

Feature vector

Class value

$$f(x, w, b) = sign(w \circ x + b)$$

$$= sign\left(\sum w[i]x[i] + b\right)$$

Linear classifiers: how would you separate these two groups of training examples with a straight line?

Feature vector

Class value

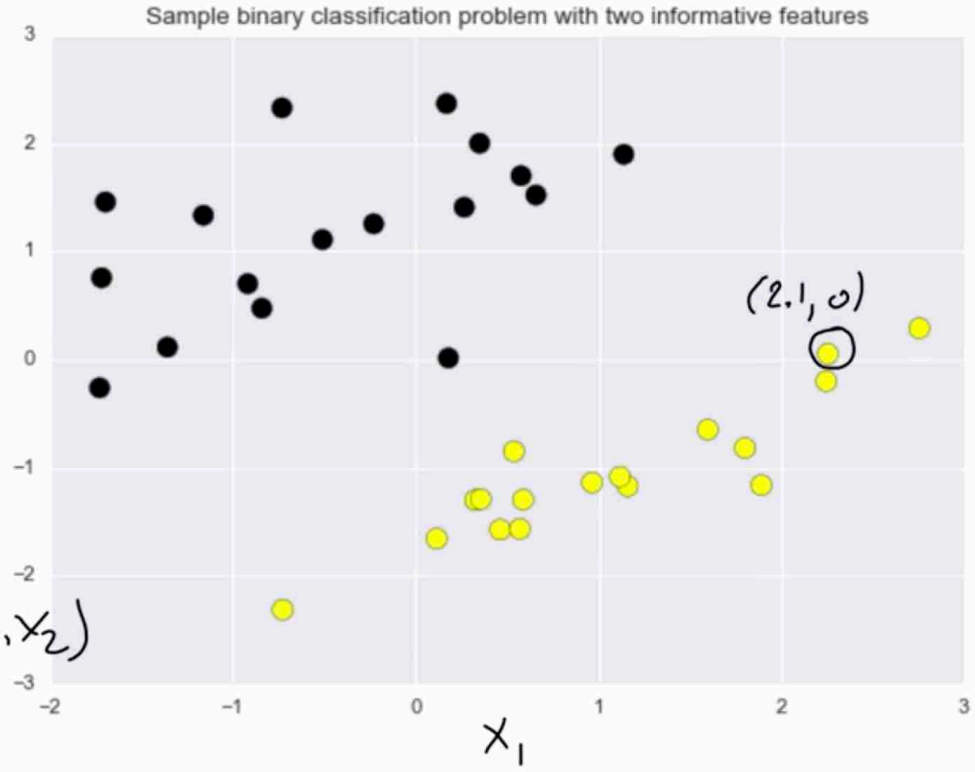
+1
−1

$$f(x, w, b) = sign(w \circ x + b)$$

$$= sign\left(\sum w[i]x[i] + b\right)$$

$(w_1, w_2) \circ (x_1, x_2)$

$= w_1 x_1 + w_2 x_2$



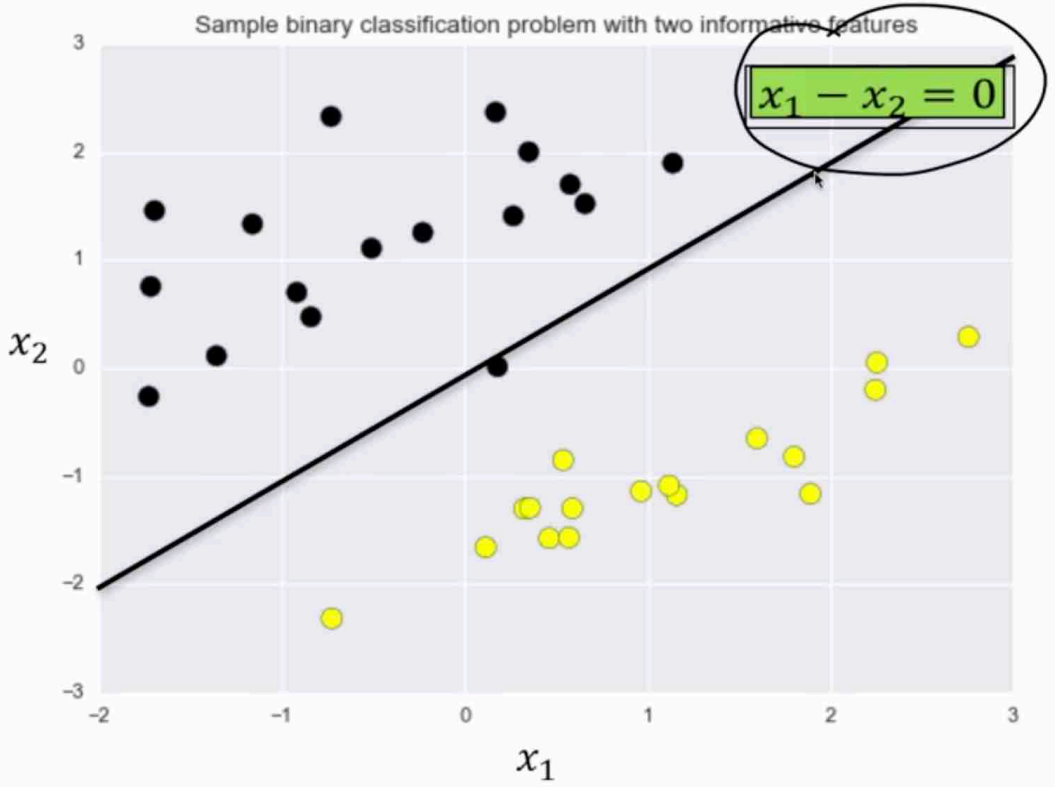
Linear classifiers: how would you separate these two groups of training examples with a line?

Feature vector

Class value

$$f(x, w, b) = sign(w \circ x + b)$$

$$x_1 - x_2 = 0$$
$$w = [1, -1]$$
$$b = 0$$



Linear classifiers: how would you separate these two groups of training examples with a line?

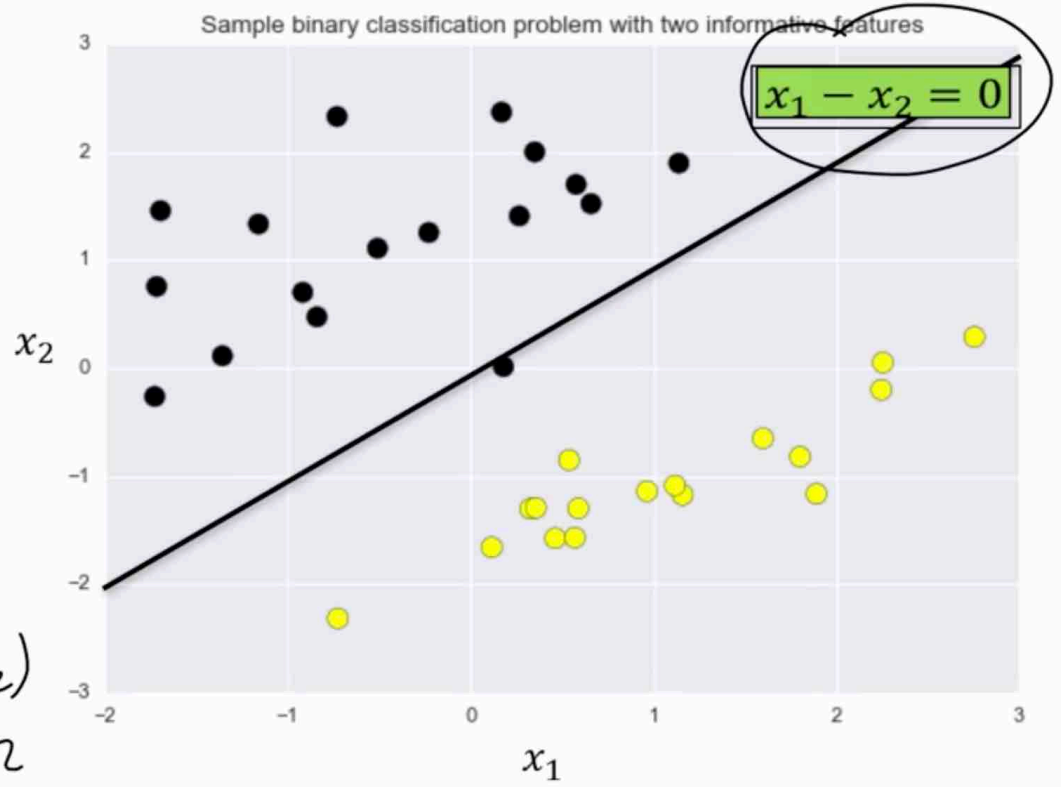
Feature vector

Class value

$$f(x, w, b) = sign(w \circ x + b)$$

$$x_1 - x_2 = 0$$
$$w = [1, -1]$$
$$b = 0$$

$(w_1, w_2) \circ (x_1, x_2)$
$= w_1 x_1 + w_2 x_2$



Linear classifiers: how would you separate these two groups of training examples with a line?

Feature vector

Class value

$x \rightarrow f \rightarrow \hat{y}$

$f(x, w, b) = sign(w \circ x + b)$

$x_1 - x_2 = 0$

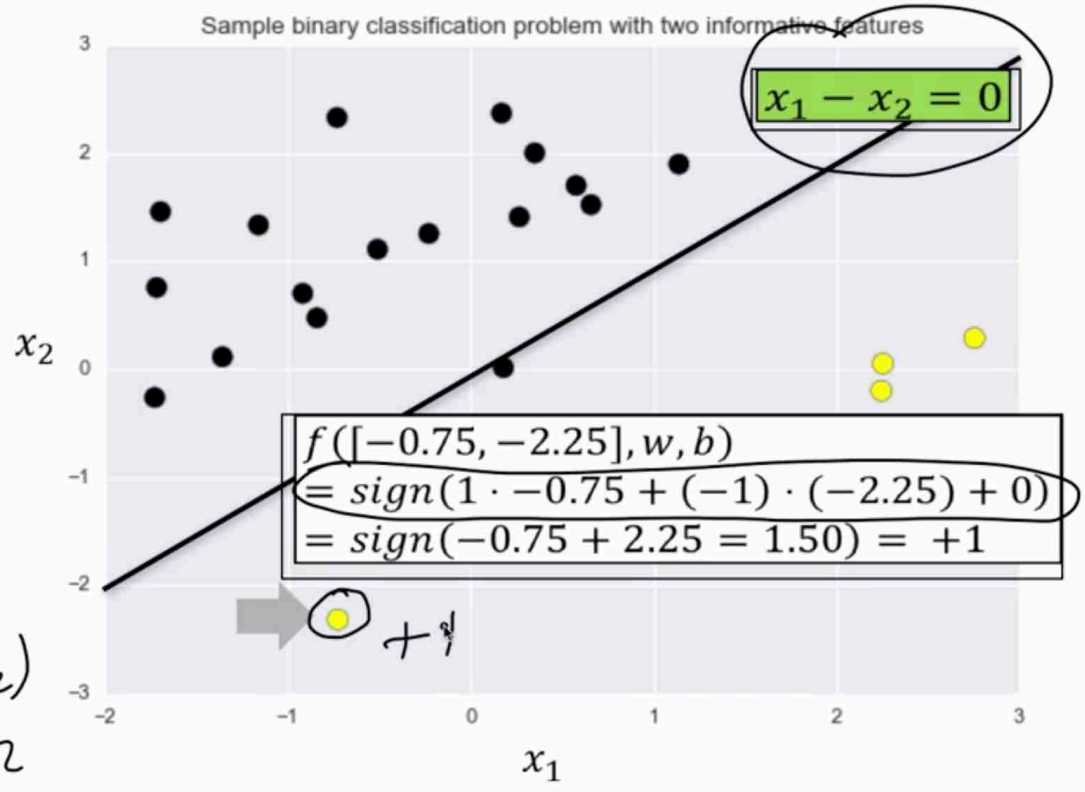
$w = [1, -1]$

$b = 0$

$(w_1, w_2) \circ (x_1, x_2)$

$= w_1 x_1 + w_2 x_2$

$= x_1 - x_2$



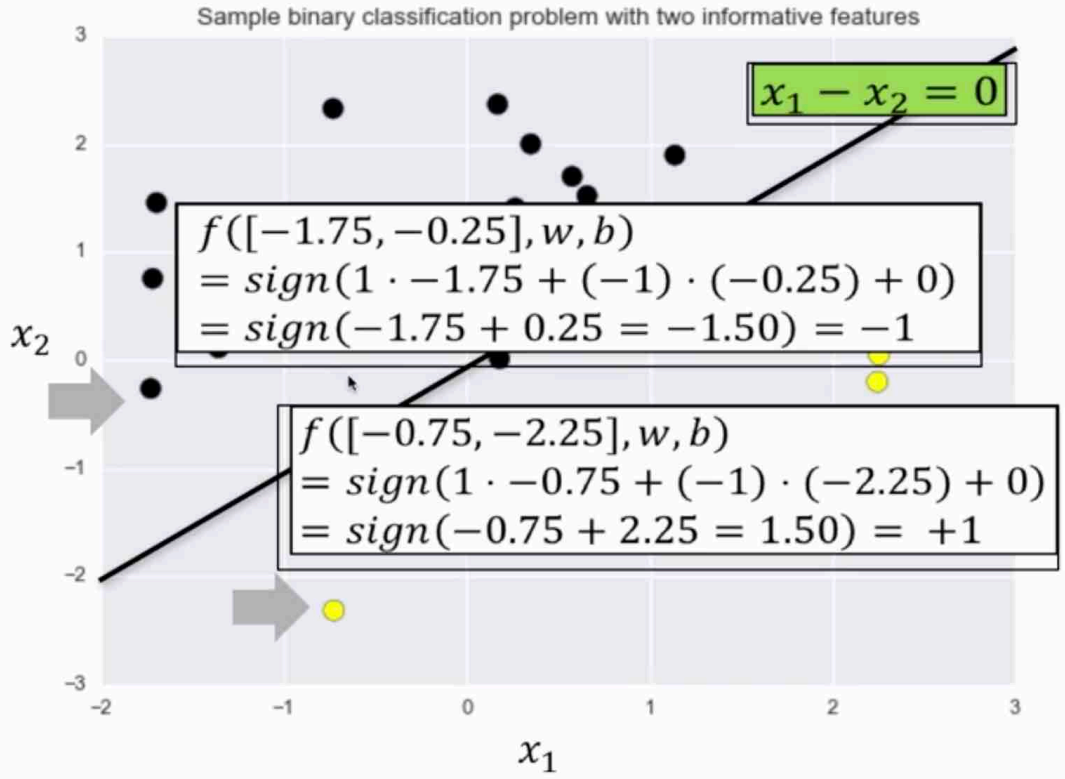
Linear classifiers: how would you separate these two groups of training examples with a line?

Feature vector

Class value

$$f(x, w, b) = sign(w \circ x + b)$$

$$x_1 - x_2 = 0$$
$$w = [1, -1]$$
$$b = 0$$



Linear classifiers: how would you separate these two groups of training examples with a line?

Feature vector

Class value

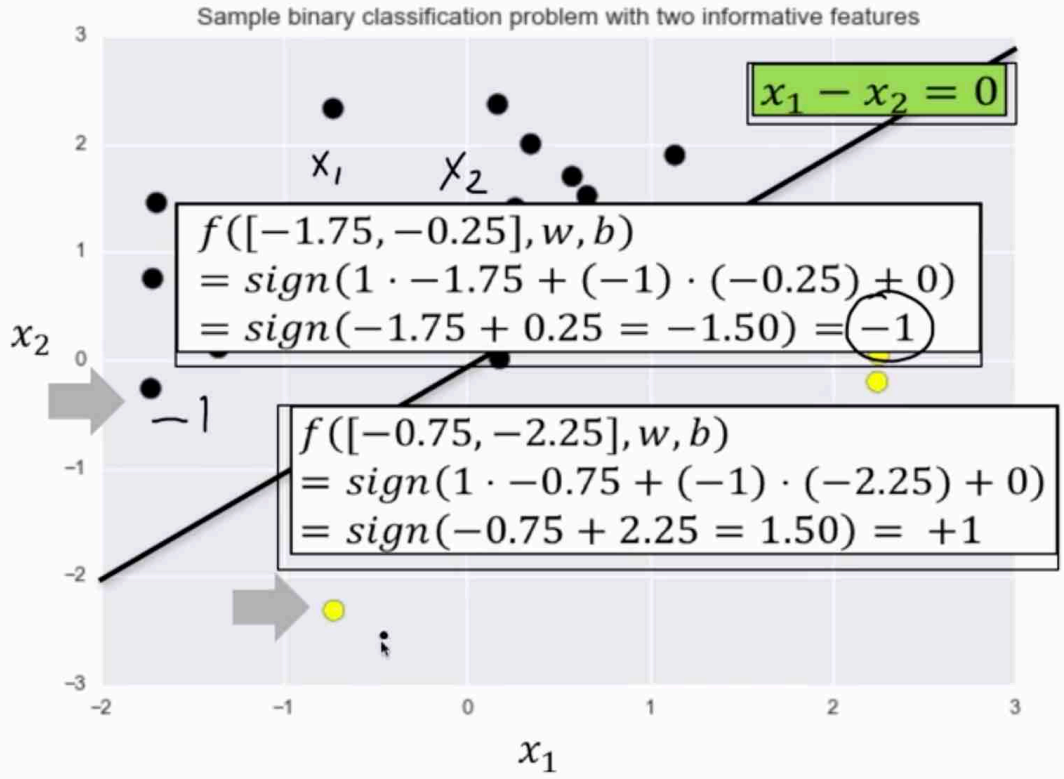
$x \rightarrow f \rightarrow \hat{y}$

$$f(x, w, b) = sign(w \circ x + b)$$

$$x_1 - x_2 = 0$$

$$w = [1, -1]$$

$$b = 0$$

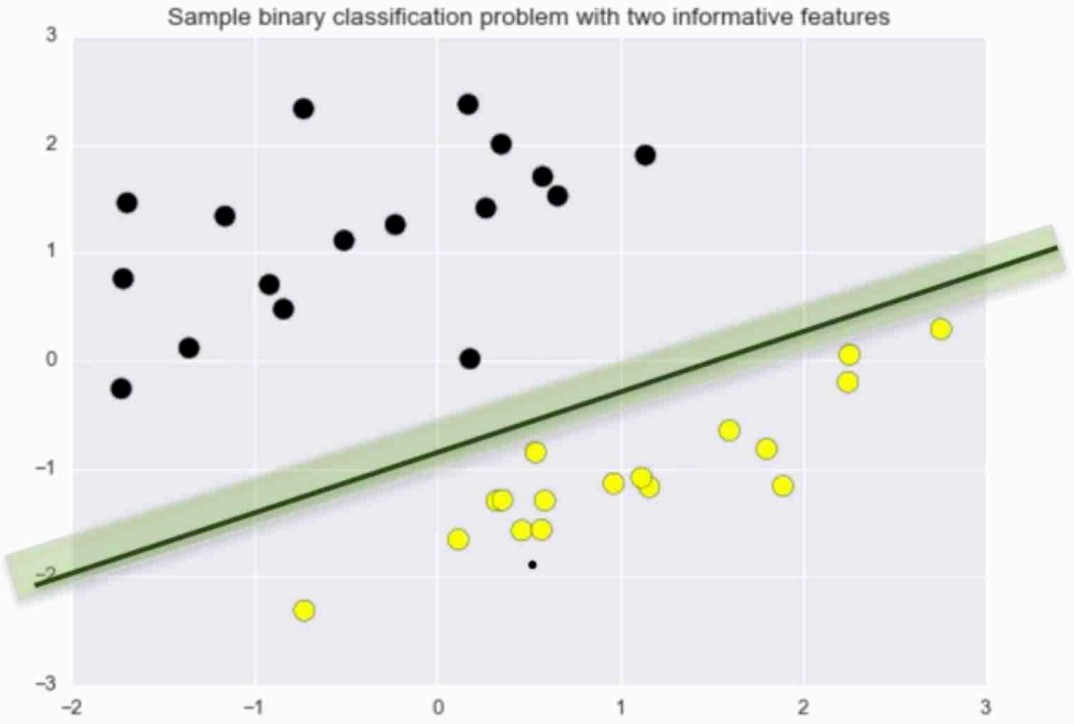


Classifier Margin

$$f(x, w, b) = sign(w \circ x + b)$$

Classifier margin

Defined as the maximum width the decision boundary area can be increased before hitting a data point.

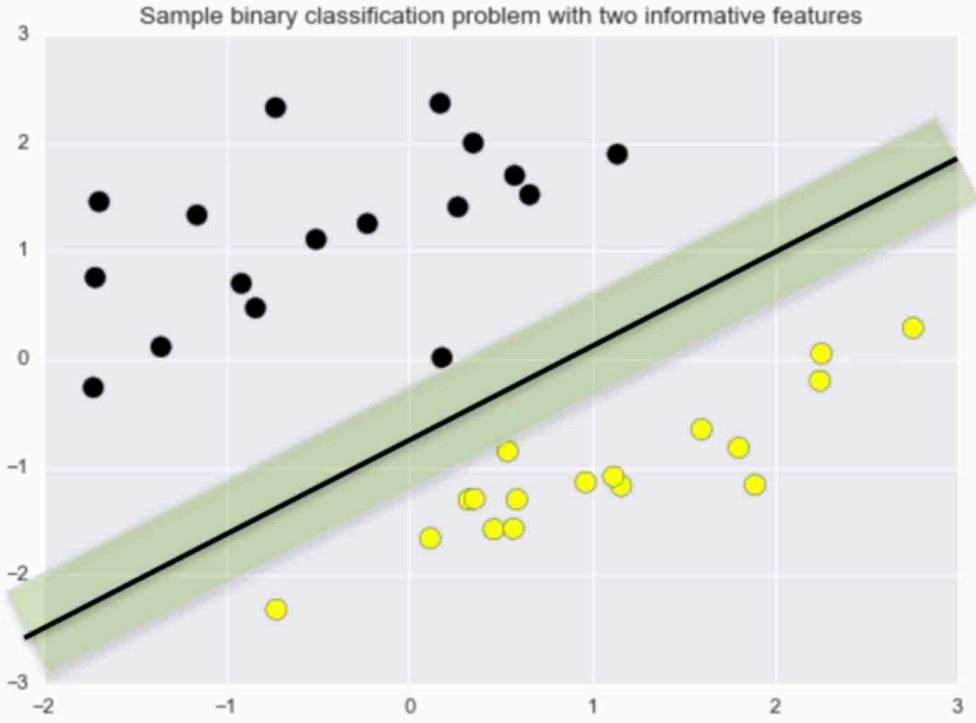


Maximum Margin Linear Classifier: Linear Support Vector Machines

$$f(x, w, b) = sign(w \circ x + b)$$

Maximum margin classifier

The linear classifier with maximum margin is a linear Support Vector Machine (LSVM).

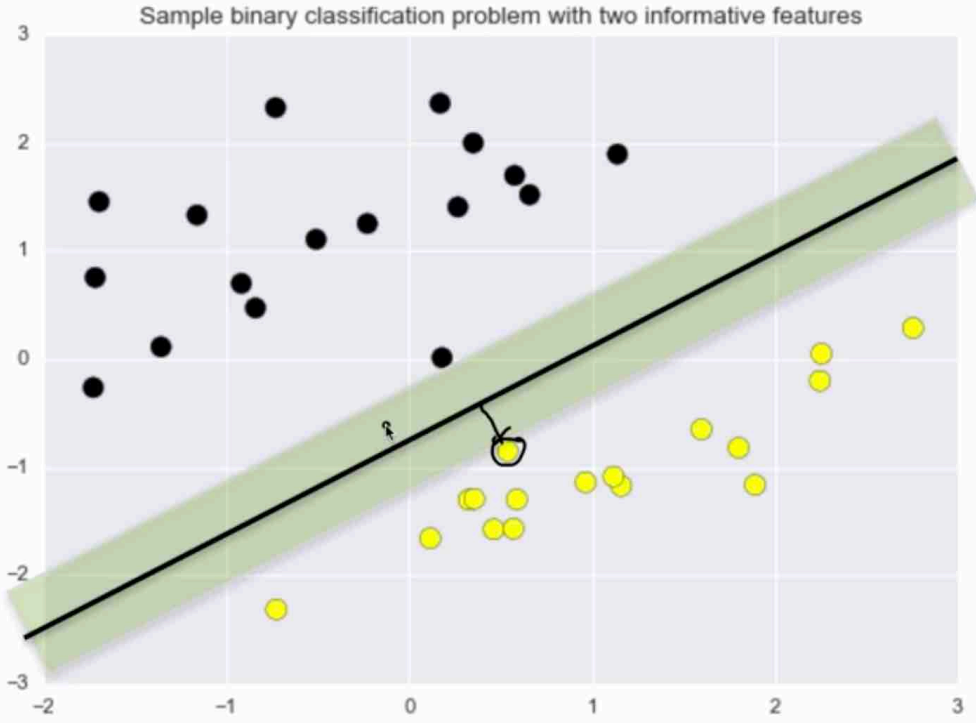


Maximum Margin Linear Classifier: Linear Support Vector Machines

$$f(x, w, b) = sign(w \circ x + b)$$

Maximum margin classifier

The linear classifier with maximum margin is a linear Support Vector Machine (LSVM).



```
.format(clf.score(X_test, y_test)))
```

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

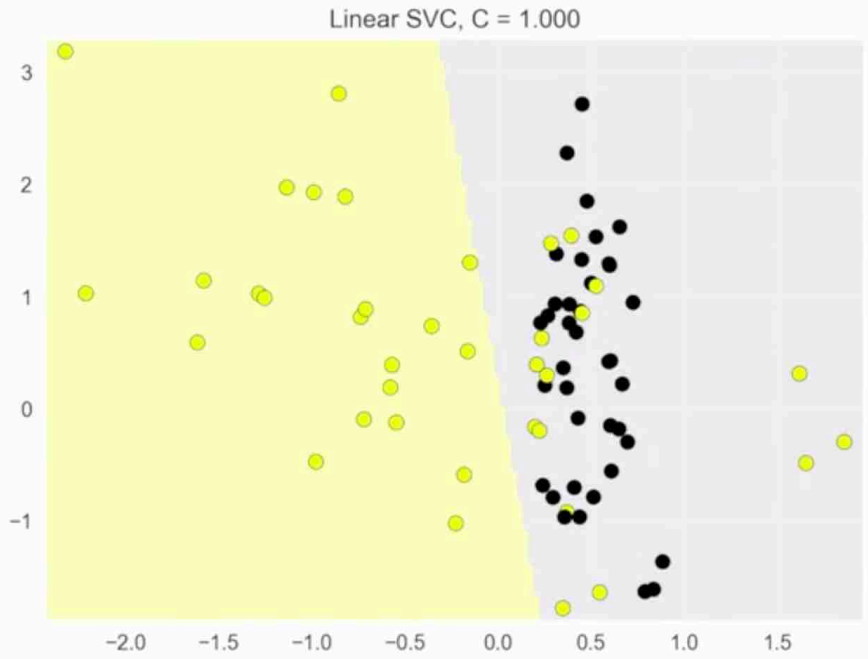
Support Vector Machines

Linear Support Vector Machine

In [ ]:

```
from sklearn.svm import SVC
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, title, subaxes)
```

Figure 6



In [ ]:

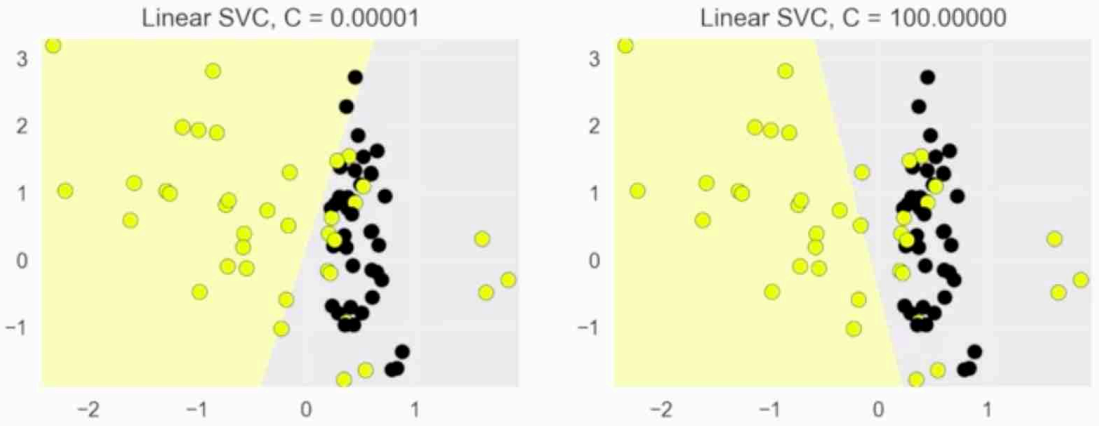
11:44 / 13:43

Regularization for SVMs: the C parameter

- The strength of regularization is determined by C
- Larger values of C: less regularization
  - *Fit the training data as well as possible*
  - *Each individual data point is important to classify correctly*
- Smaller values of C: more regularization
  - *More tolerant of errors on individual data points*

In [21]:
```
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=(9, 3))

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)
```

Figure 7



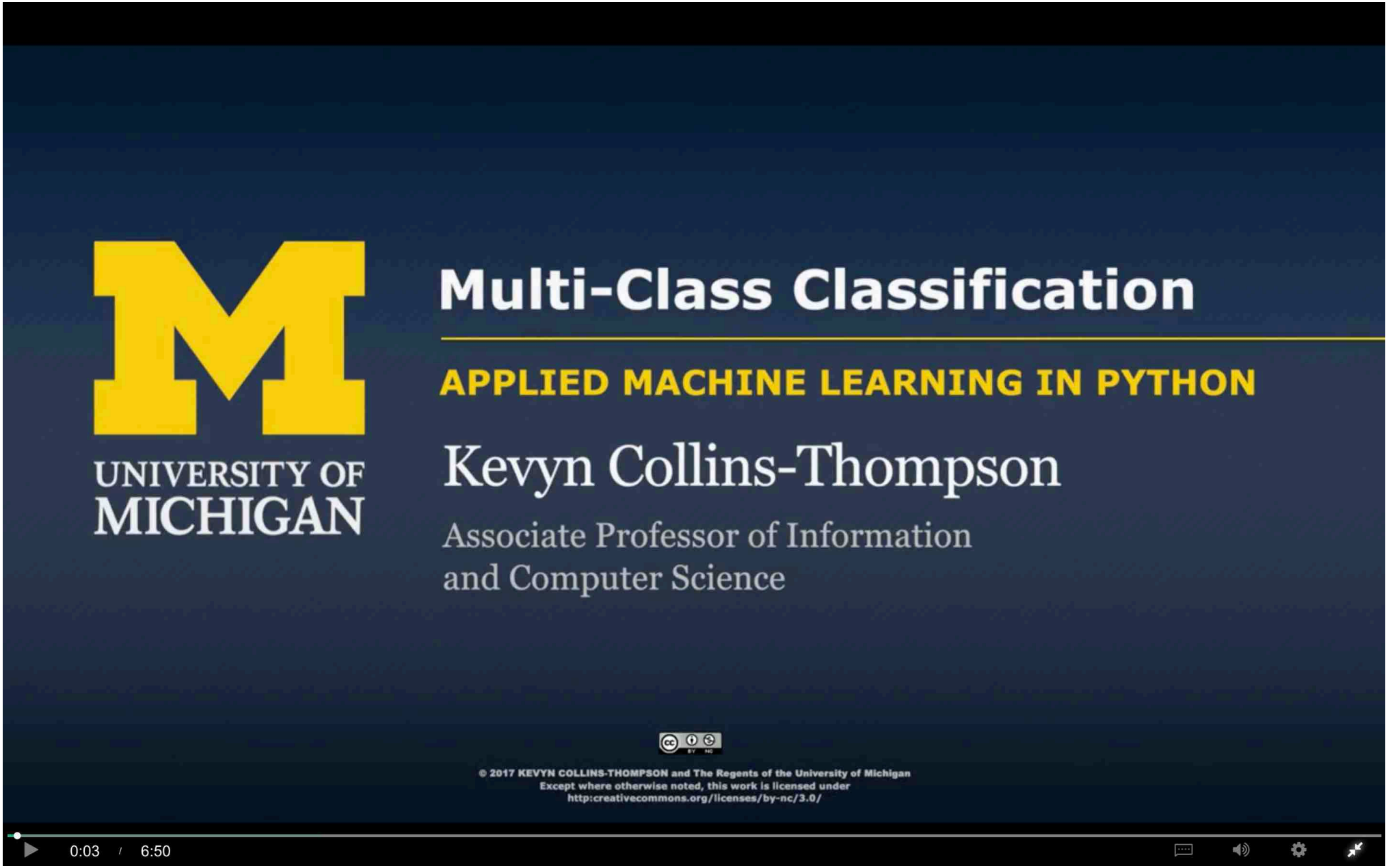
Linear Models: Pros and Cons

Pros:

- Simple and easy to train.
- Fast prediction.
- Scales well to very large datasets.
- Works well with sparse data.
- Reasons for prediction are relatively easy to interpret.

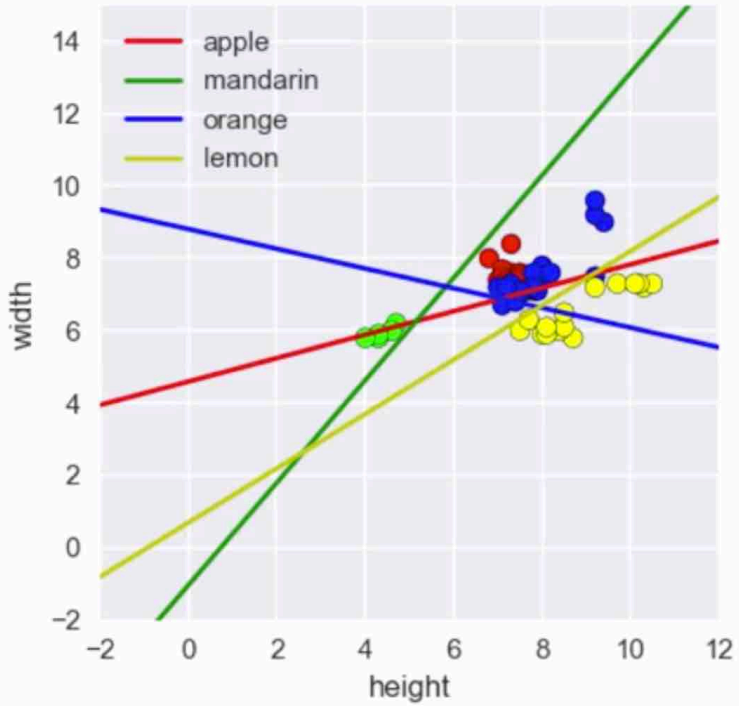
Cons:

- For lower-dimensional data, other models may have superior generalization performance.
- For classification, data may not be linearly separable (more on this in SVMs with non-linear kernels)


UNIVERSITY OF MICHIGAN
Multi-Class Classification
APPLIED MACHINE LEARNING IN PYTHON
Kevyn Collins-Thompson
Associate Professor of Information and Computer Science
© 2017 KEVYN COLLINS-THOMPSON and The Regents of the University of Michigan
Except where otherwise noted, this work is licensed under
http:creativecommons.org/licenses/by-nc/3.0/
0:03 / 6:50

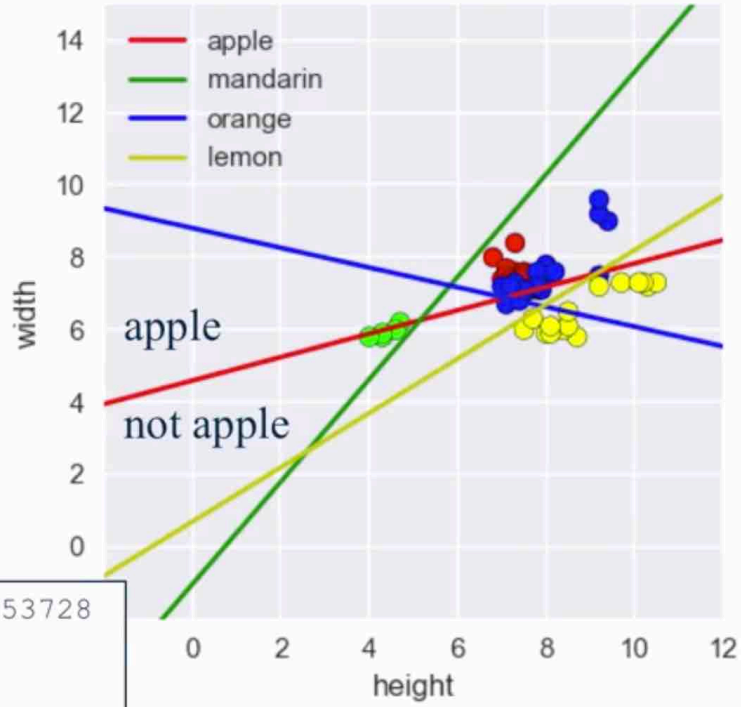
Multi-class Classification with Linear Models

```
clf = LinearSVC(C=5, random_state = 67)
clf.fit(X_train, y_train)

print(clf.coef_)

[[-0.23401135  0.72246132]
 [-1.63231901  1.15222281]
 [ 0.0849835   0.31186707]
 [ 1.26189663 -1.68097   ]]

print(clf.intercept_)
[-3.31753728  1.19645936 -2.7468353  1.16107418]
```



Multi-class Classification with Linear Models

```
clf = LinearSVC(C=5, random_state = 67)
clf.fit(X_train, y_train)

print(clf.coef_)

[[-0.23401135  0.72246132]
 [-1.63231901  1.15222281]
 [ 0.0849835   0.31186707]
 [ 1.26189663 -1.68097   ]]

print(clf.intercept_)
[-3.31753728  1.19645936 -2.7468353  1.16107418]
```

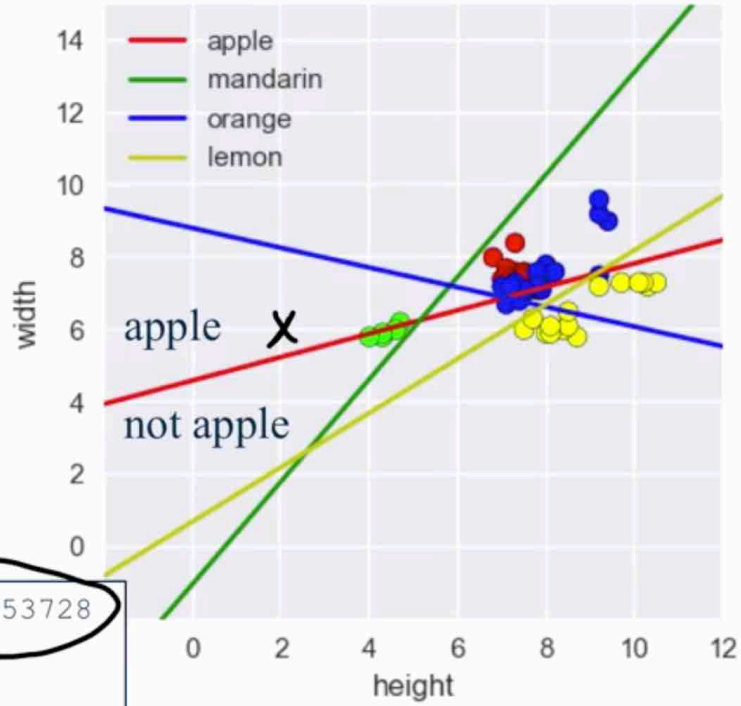


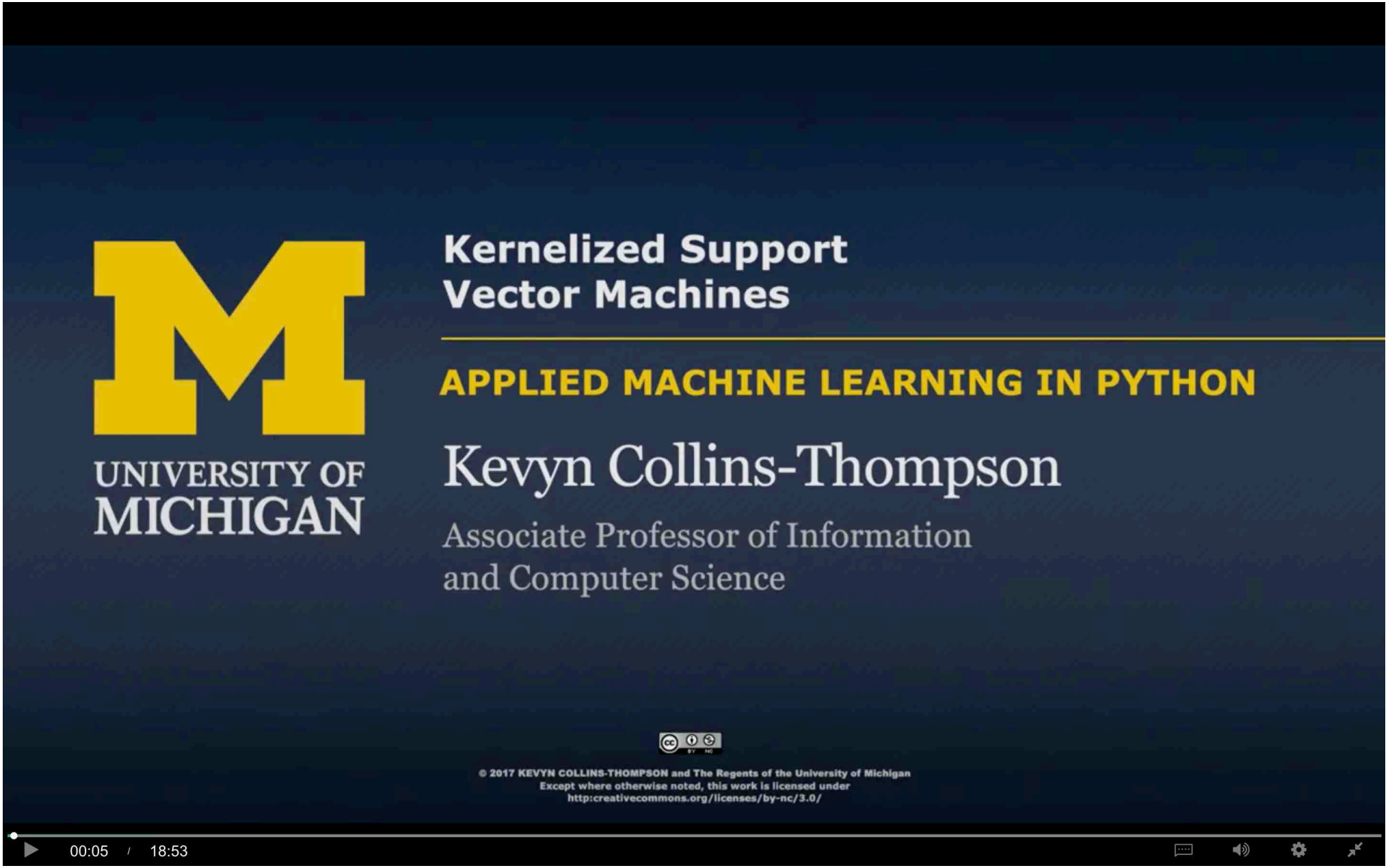

```
y_apple = -0.23401135 * height + 0.72246132 * width - 3.31753728

height=2, width=6:  y_apple = + 0.549 (>= 0: predict apple)
height=2, width=2:  y_apple = - 2.340 (< 0:  predict other)
```

Multi-class Classification with Linear Models

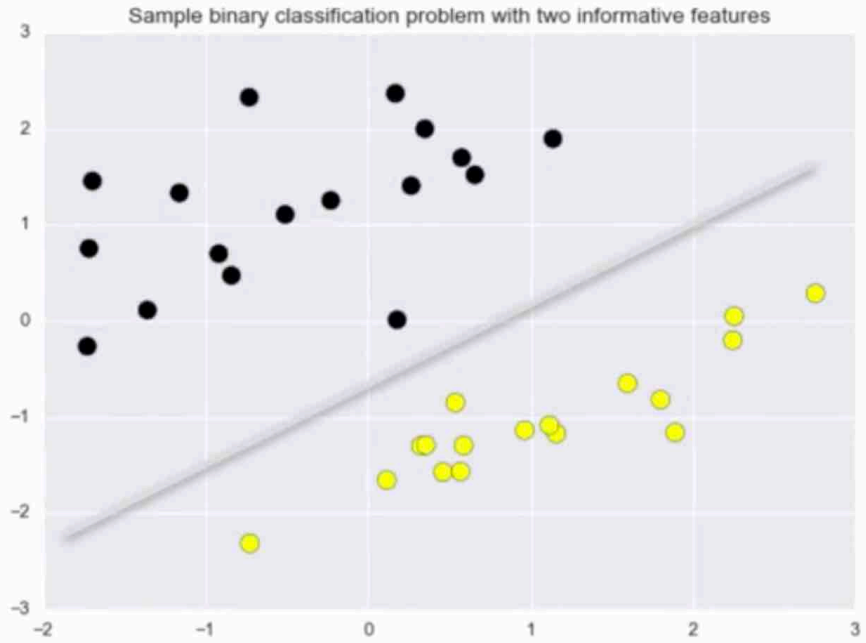
```
clf = LinearSVC(C=5, random_state = 67)
clf.fit(X_train, y_train)

print(clf.coef_)

[[-0.23401135  0.72246132]
 [-1.63231901  1.15222281]
 [ 0.0849835   0.31186707]
 [ 1.26189663 -1.68097   ]]

print(clf.intercept_)
[-3.31753728  1.19645936 -2.7468353   1.16107418]
```

```
y_apple = -0.23401135 * height + 0.72246132 * width - 3.31753728

height=2, width=6:  y_apple = + 0.549 (>= 0: predict apple)
height=2, width=2:  y_apple = - 2.340 (< 0:  predict other)
```



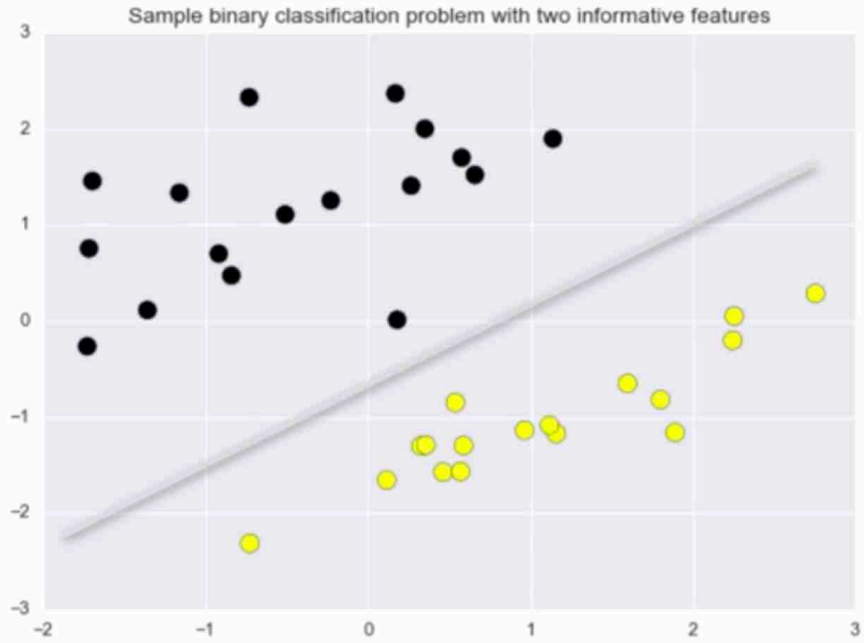

Kernelized Support Vector Machines
APPLIED MACHINE LEARNING IN PYTHON
UNIVERSITY OF MICHIGAN
Kevyn Collins-Thompson
Associate Professor of Information and Computer Science
© 2017 KEVYN COLLINS-THOMPSON and The Regents of the University of Michigan
Except where otherwise noted, this work is licensed under
http:creativecommons.org/licenses/by-nc/3.0/
00:05 / 18:53

We saw how linear support vector classifiers could effectively find a decision boundary with maximum margin

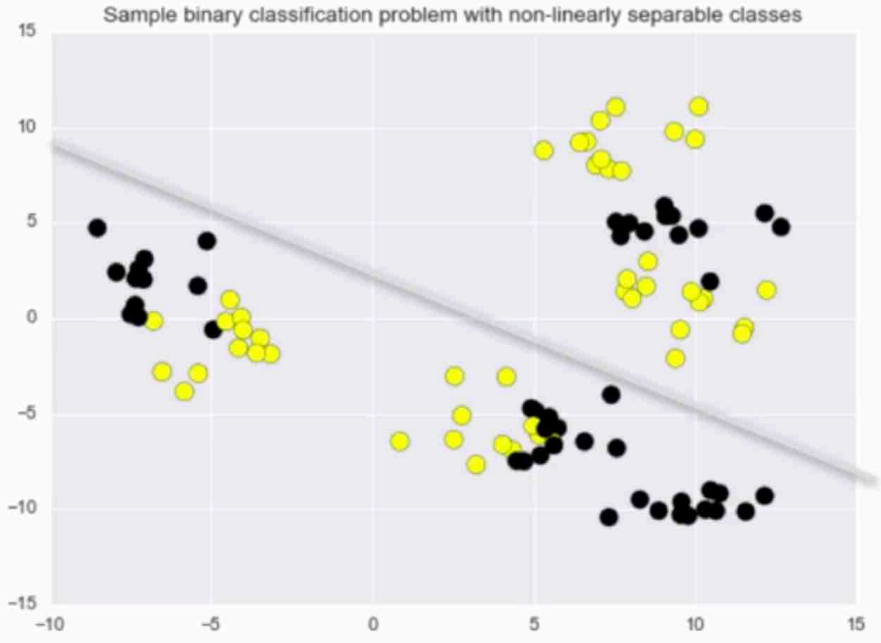


Easy for a linear classifier

But what about more complex binary classification problems?

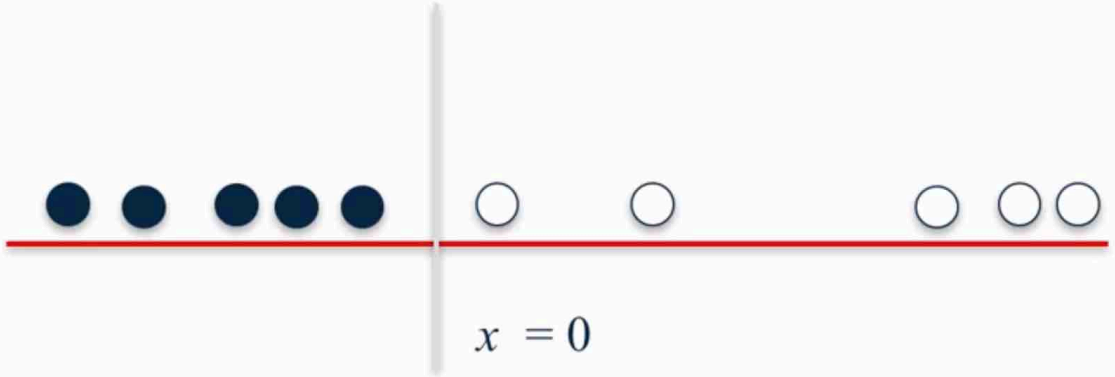


Easy for a linear classifier

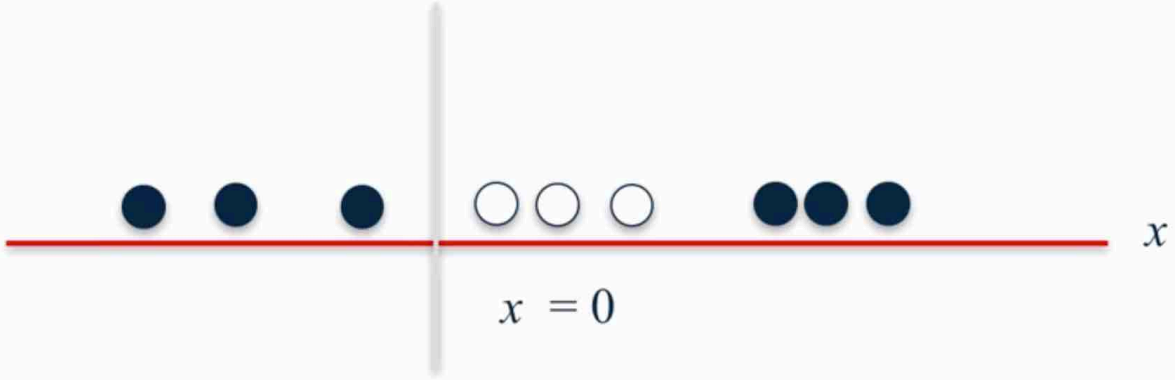


Difficult/impossible for a linear classifier

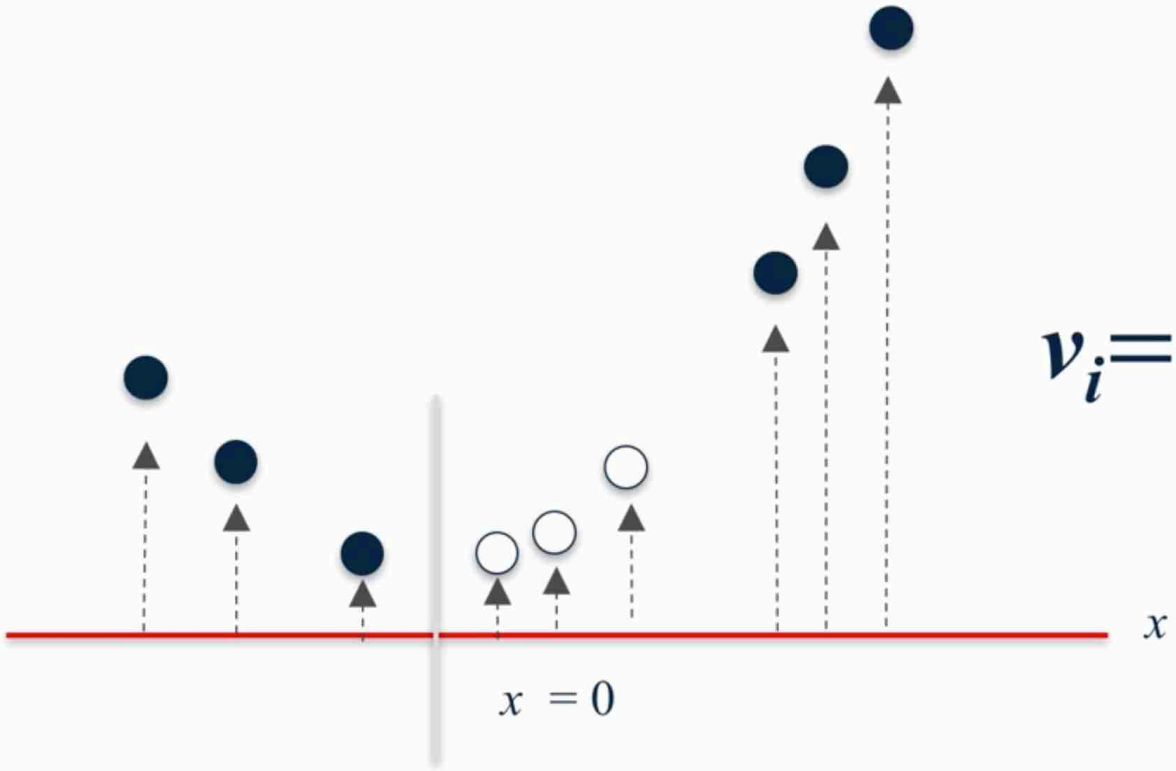
A simple 1-dimensional classification problem for a linear classifier



A more perplexing 1-d classification problem for a linear classifier

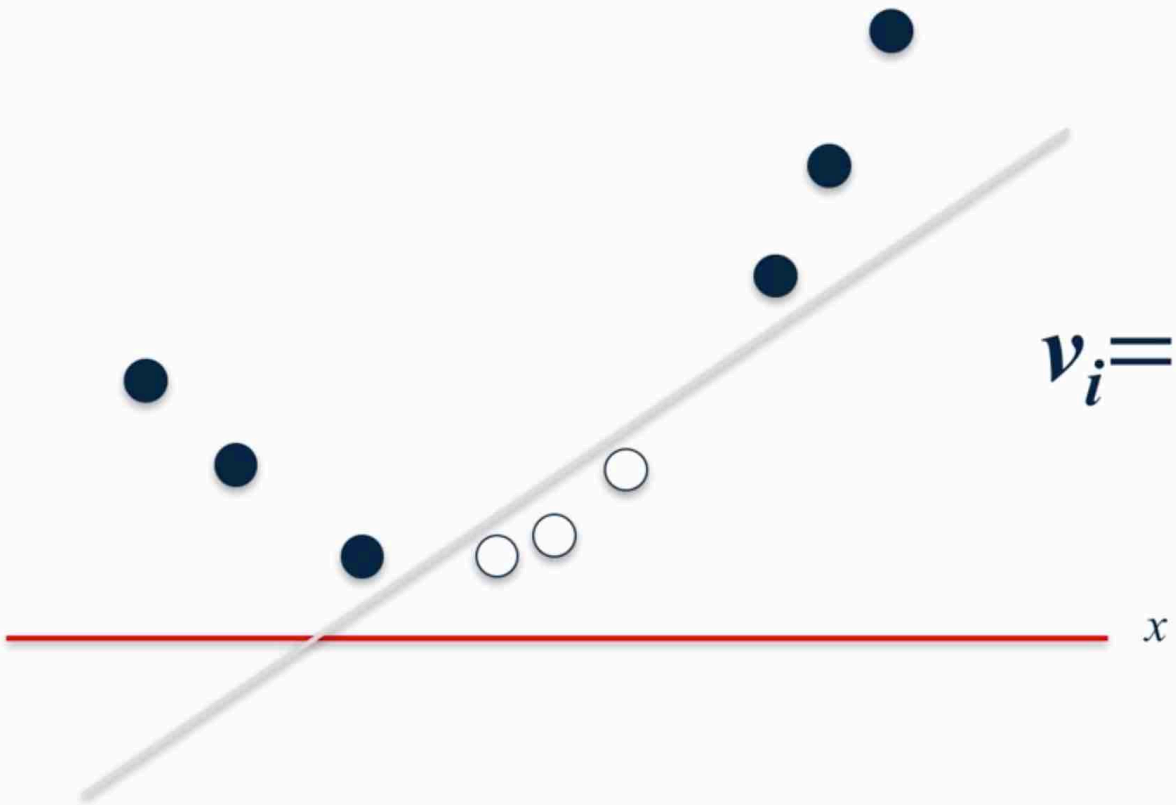


Let's transform the data by adding a second dimension/feature (set to the squared value of the first feature)



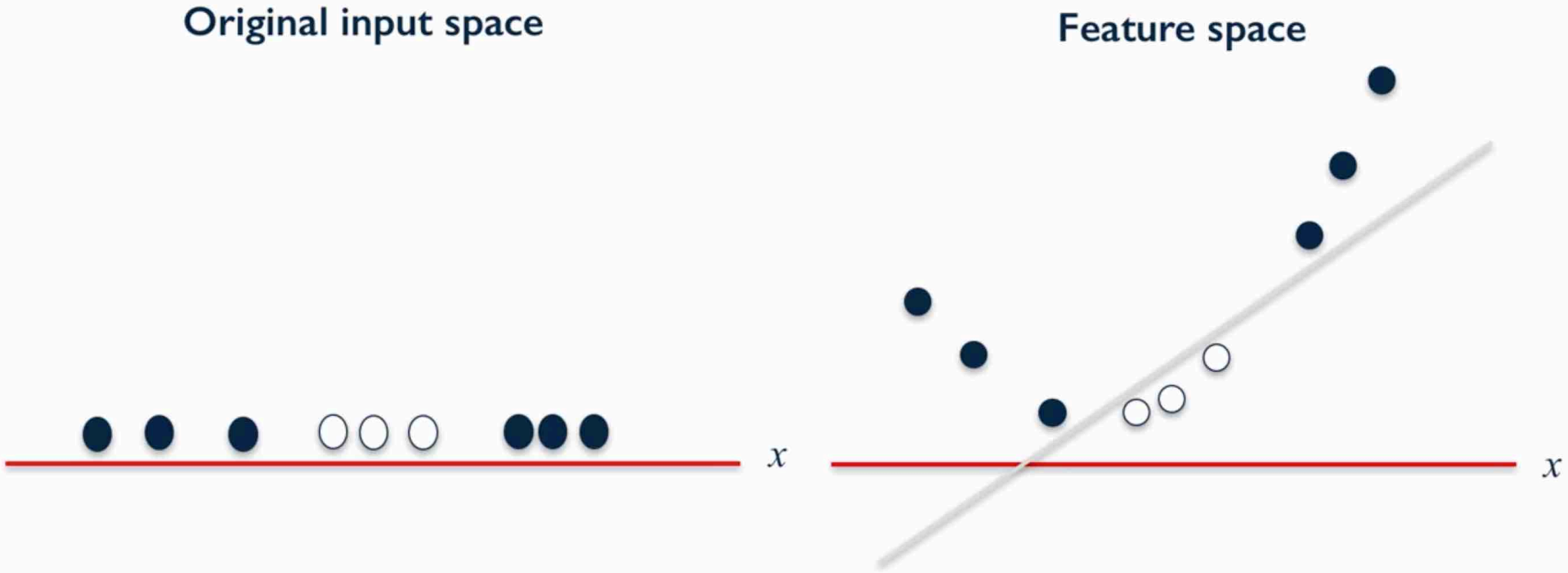
$$v_i = (x_i, x_i^2)$$

The data transformation makes it possible to solve this with a linear classifier

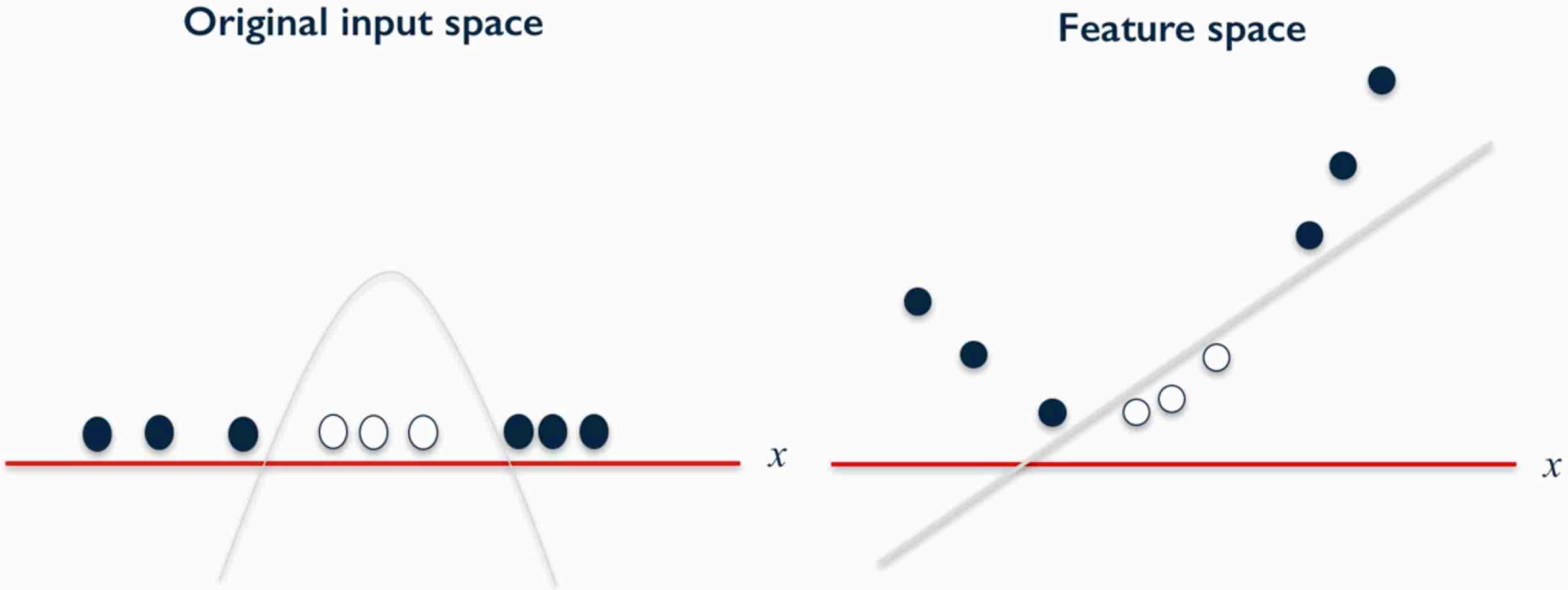


$$v_i = (x_i, x_i^2)$$

What does the linear decision boundary in feature space correspond to in the original input space?

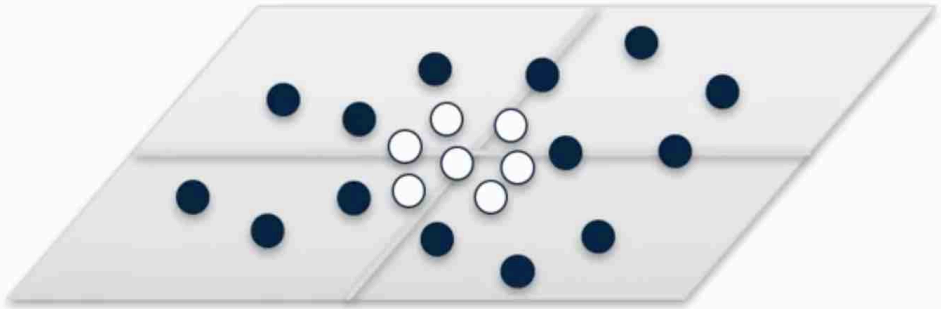


What does the linear decision boundary correspond to in the original input space?



Example of mapping a 2D classification problem to a 3D feature space to make it linearly separable

Original input space

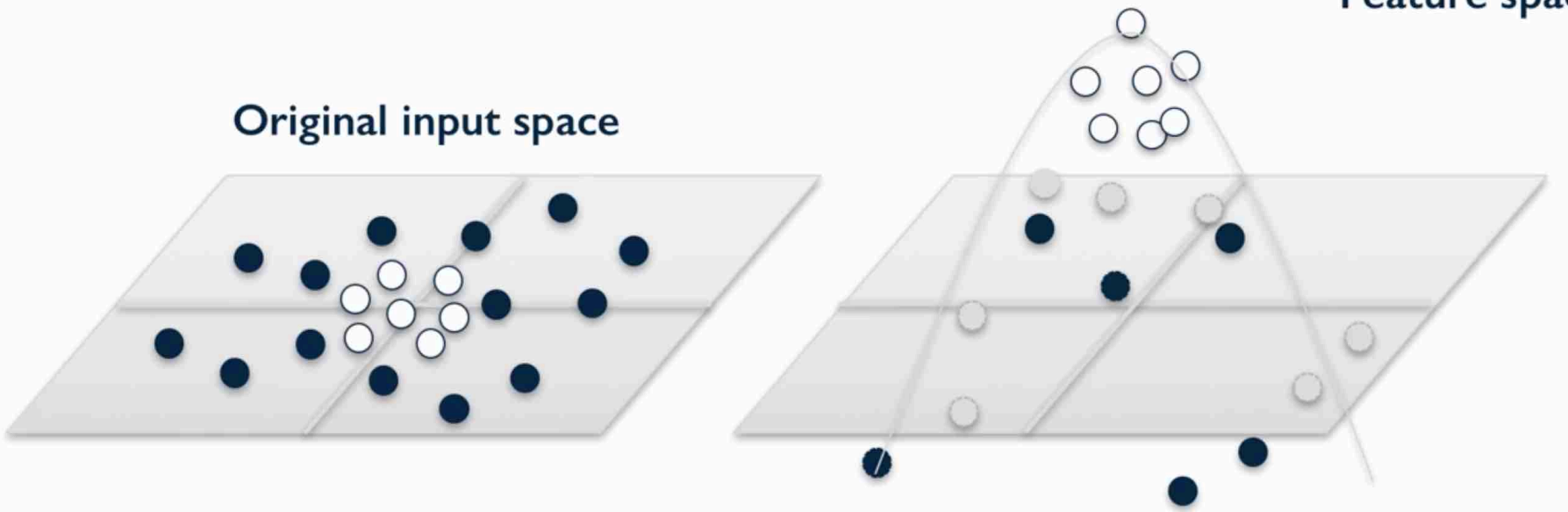


$$x_i = (x_0, x_1)$$

Example of mapping a 2D classification problem to a 3D feature space to make it linearly separable

Feature space

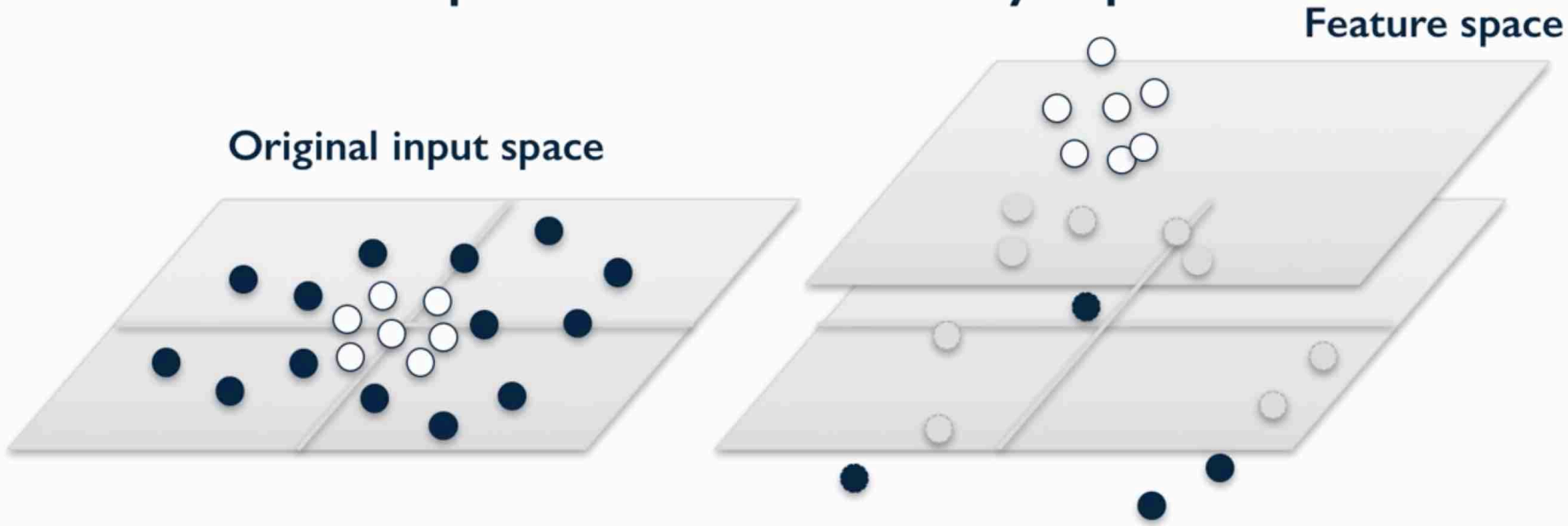
Original input space



$$x_i = (x_0, x_1)$$

$$v_i = (x_0, x_1, 1 - (x_0^2 + x_1^2))$$

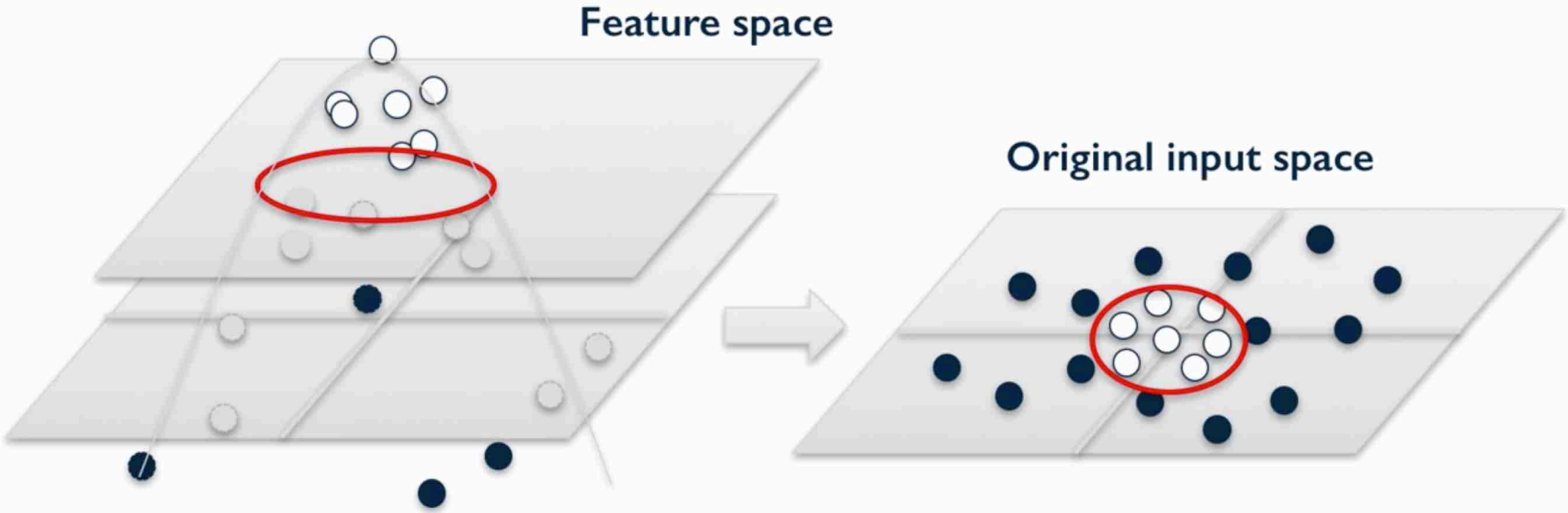
Example of mapping a 2D classification problem to a 3D feature space to make it linearly separable



$$x_i = (x_0, x_1)$$

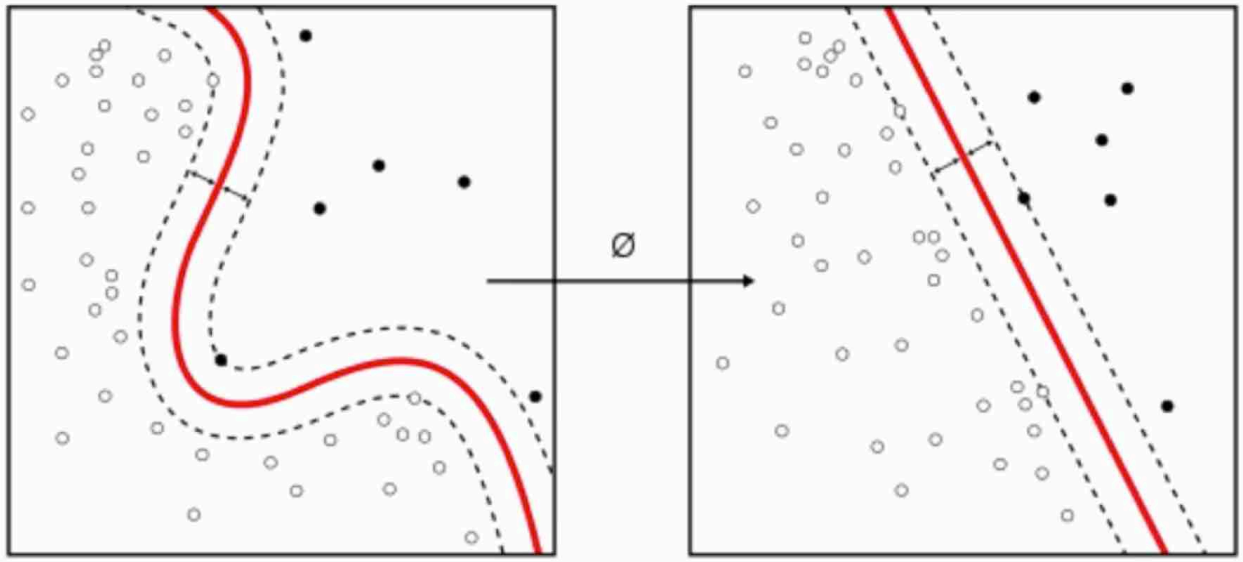
$$v_i = (x_0, x_1, 1 - (x_0^2 + x_1^2))$$

Example of mapping a 2D classification problem to a 3D feature space to make it linearly separable



$$\boldsymbol{v_i} = (x_0, x_1, 1 - (x_0^2 + x_1^2)) \qquad \boldsymbol{x_i} = (x_0, x_1)$$

Transforming the data can make it much easier for a linear classifier.



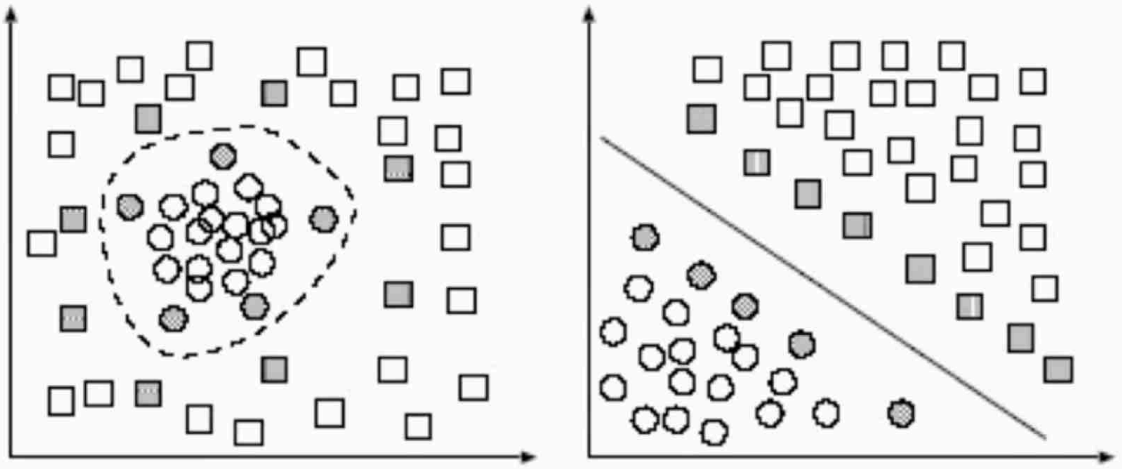
Original input space

Feature space

Source: Wikipedia "Kernel Machine" article.
https://commons.wikimedia.org/w/index.php?curid=47868867

Radial Basis Function Kernel

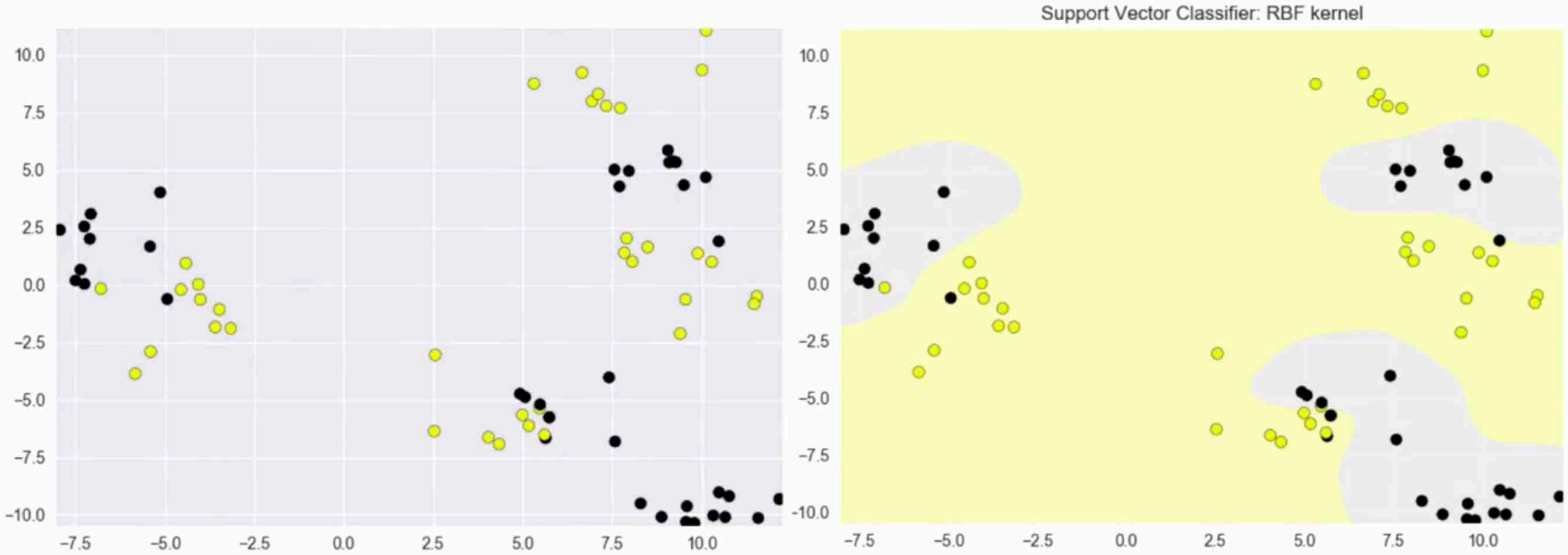
$$K(\boldsymbol{x}, \boldsymbol{x}') = \exp\ [-\gamma \cdot \|\boldsymbol{x} - \boldsymbol{x}'\|^2]$$



Original input space Feature space

A kernel is a similarity measure (modified dot product) between data points

Applying the SVM with RBF kernel



Classification

In [*]:
```
from sklearn.svm import SVC
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)

plot_class_regions_for_classifier(SVC().fit(X_train, y_train),
                                  X_train, y_train, None, None,
                                  'Support Vector Classifier: RBF kernel')

plot_class_regions_for_classifier(SVC(kernel = 'poly', degree = 3)
                                  .fit(X_train, y_train), X_train,
                                  y_train, None, None,
                                  'Support Vector Classifier: \
Polynomial kernel, degree = 3')
```

In [ ]:

Figure 9

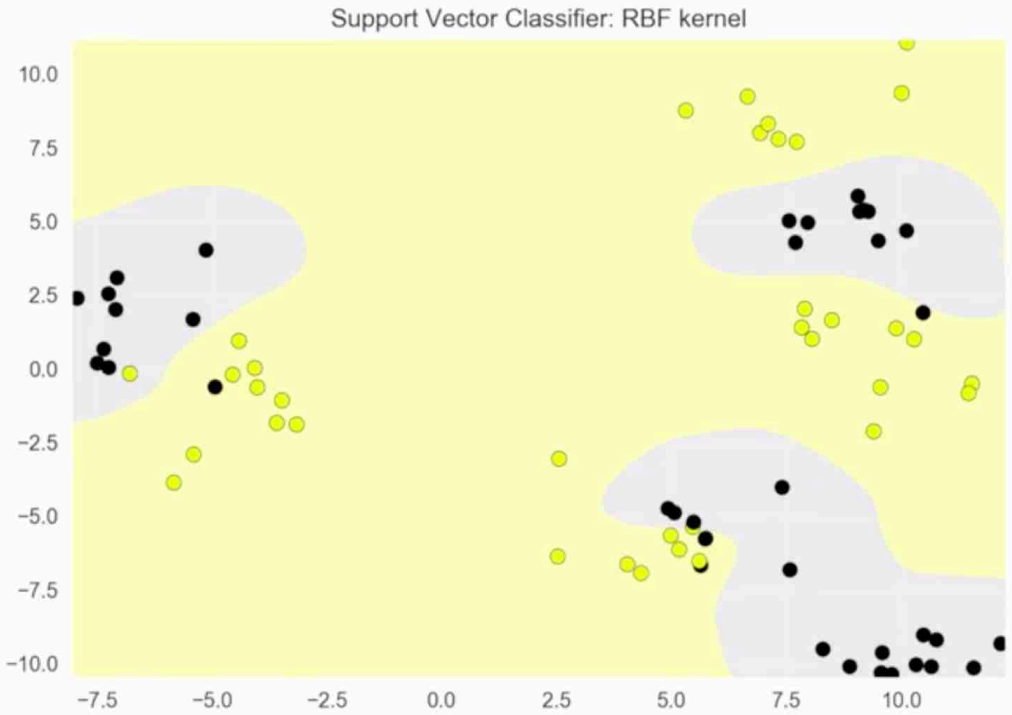


Figure 10

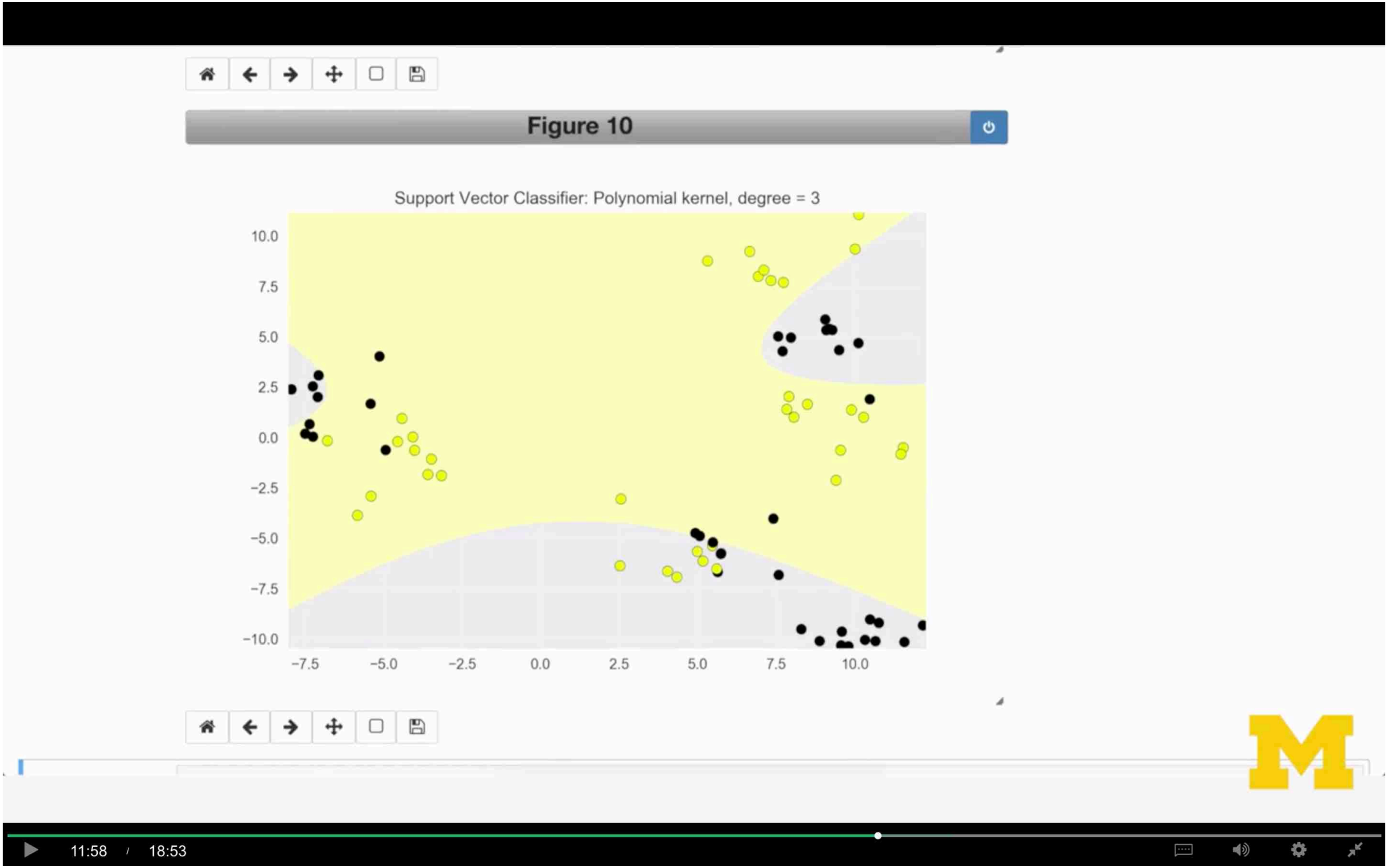


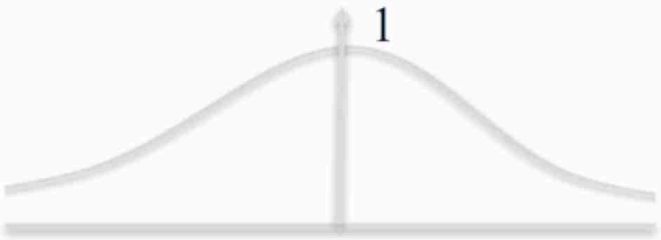
Figure 10
Support Vector Classifier: Polynomial kernel, degree = 3
10.0
7.5
5.0
2.5
0.0
−2.5
−5.0
−7.5
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0

Radial Basis Function kernel: Gamma Parameter

$$K(\boldsymbol{x}, \boldsymbol{x}') = \exp\ [-\gamma \cdot \|\boldsymbol{x} - \boldsymbol{x}'\|^2]$$

gamma (γ): kernel width parameter

small gamma (0.01)

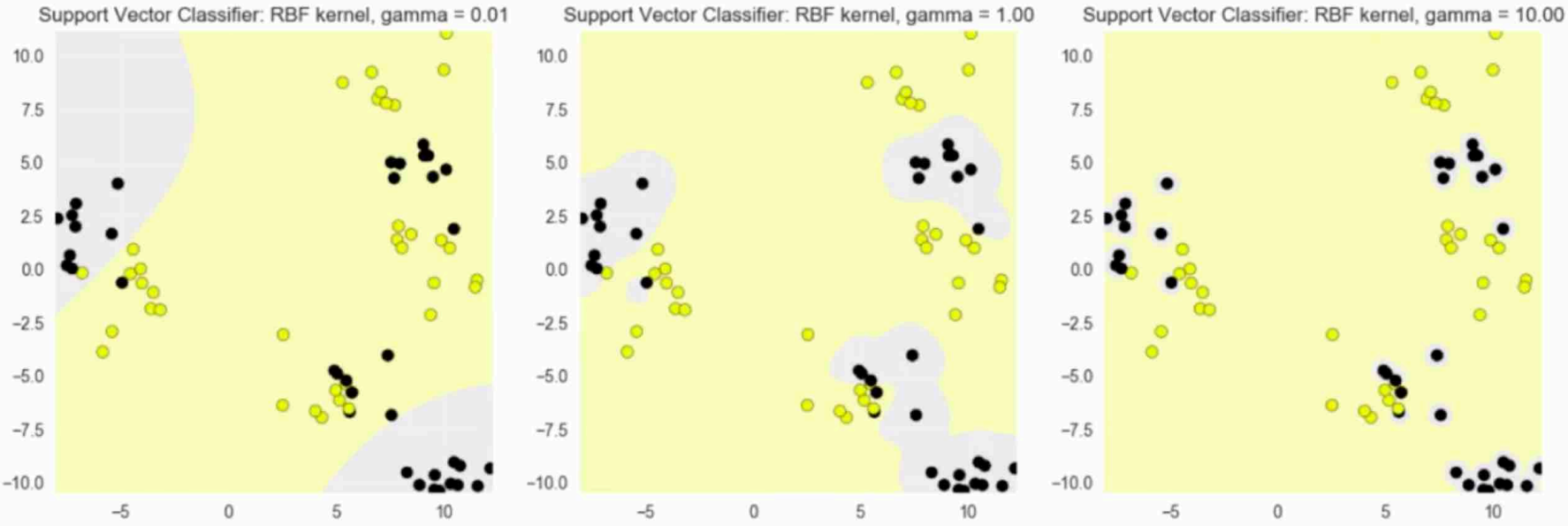


large gamma (10)

$\|\boldsymbol{x} - \boldsymbol{x}'\|^2$

Squared distance between x and x'

The effect of the RBF gamma parameter on decision boundaries



gamma = 0.01

gamma = 1.0

gamma = 10

Support Vector Machine with RBF kernel: gamma parameter

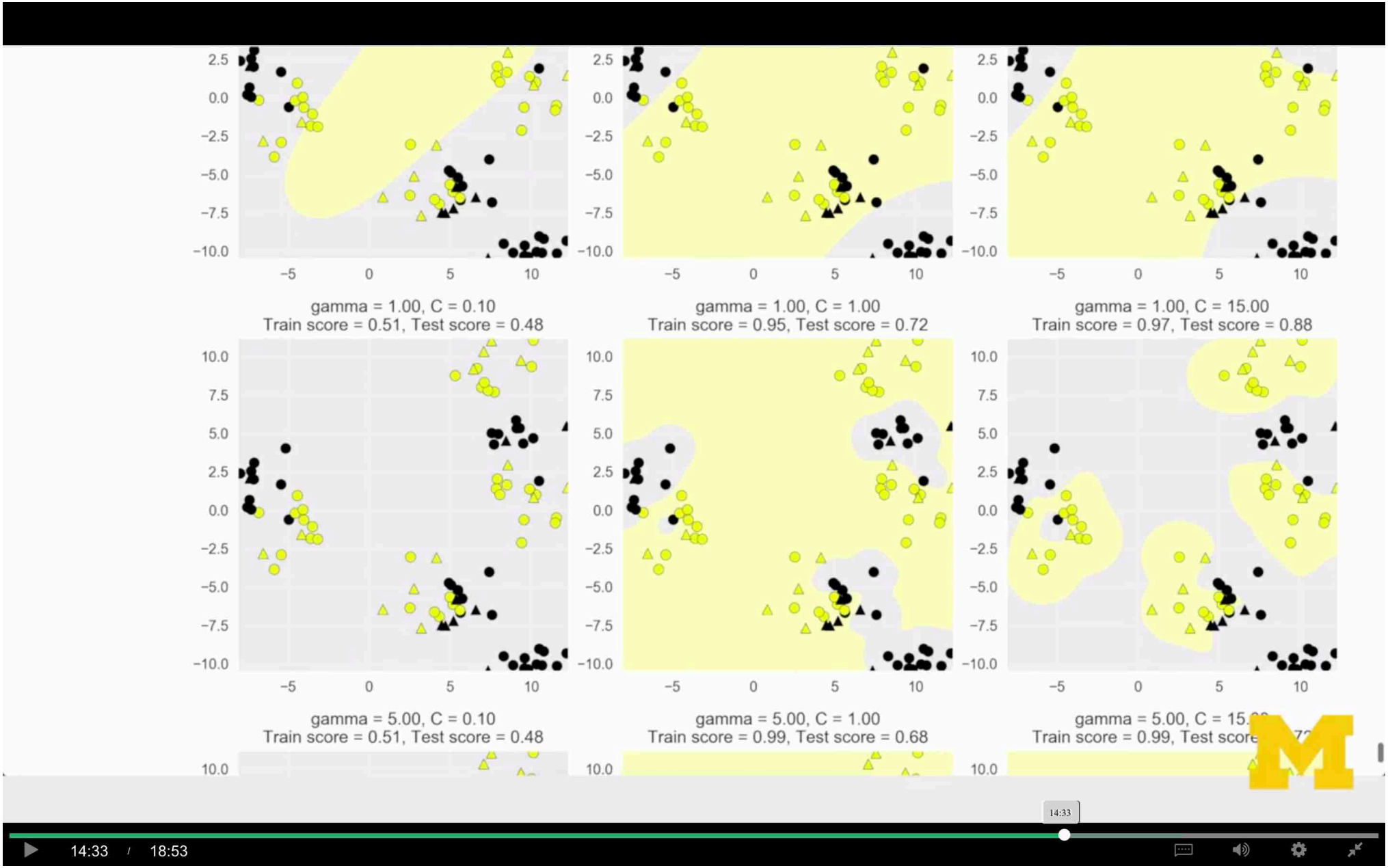
In [*]:
```
from adspy_shared_utilities import plot_class_regions_for_classifier

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

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: \n\
RBF kernel, gamma = {:.2f}'.format(this_gamma)
    plot_class_regions_for_classifier_subplot(clf, X_train, y_train,
                                              None, None, title, subplot)
    plt.tight_layout()
```

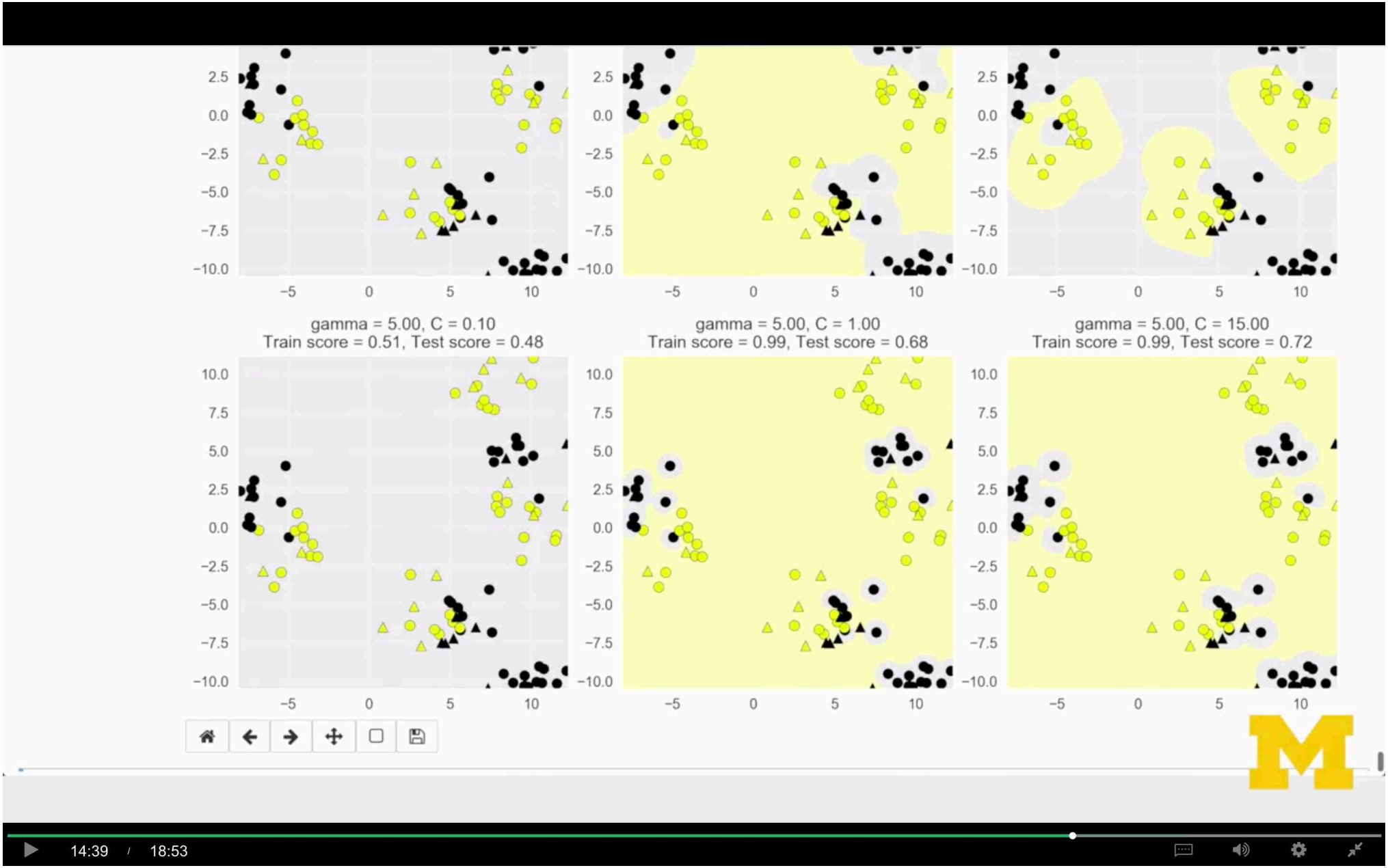
In [ ]:

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

In [ ]:

```
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, 12))

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)
```


gamma = 1.00, C = 0.10
Train score = 0.51, Test score = 0.48
gamma = 1.00, C = 1.00
Train score = 0.95, Test score = 0.72
gamma = 1.00, C = 15.00
Train score = 0.97, Test score = 0.88
gamma = 5.00, C = 0.10
Train score = 0.51, Test score = 0.48
gamma = 5.00, C = 1.00
Train score = 0.99, Test score = 0.68
gamma = 5.00, C = 15.
Train score = 0.99, Test score
14:33
14:33 / 18:53


gamma = 5.00, C = 0.10
Train score = 0.51, Test score = 0.48
gamma = 5.00, C = 1.00
Train score = 0.99, Test score = 0.68
gamma = 5.00, C = 15.00
Train score = 0.99, Test score = 0.72

14:39 / 18:53

Application of SVMs to a real dataset: unnormalized data

In [28]:
```
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
```

In [ ]:

15:12 / 18:53

```
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
```

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

In [29]:
```
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
```

In [ ]:

Kernelized Support Vector Machines: pros and cons

Pros:

- Can perform well on a range of datasets.
- Versatile: different kernel functions can be specified, or custom kernels can be defined for specific data types.
- Works well for both low- and high-dimensional data.

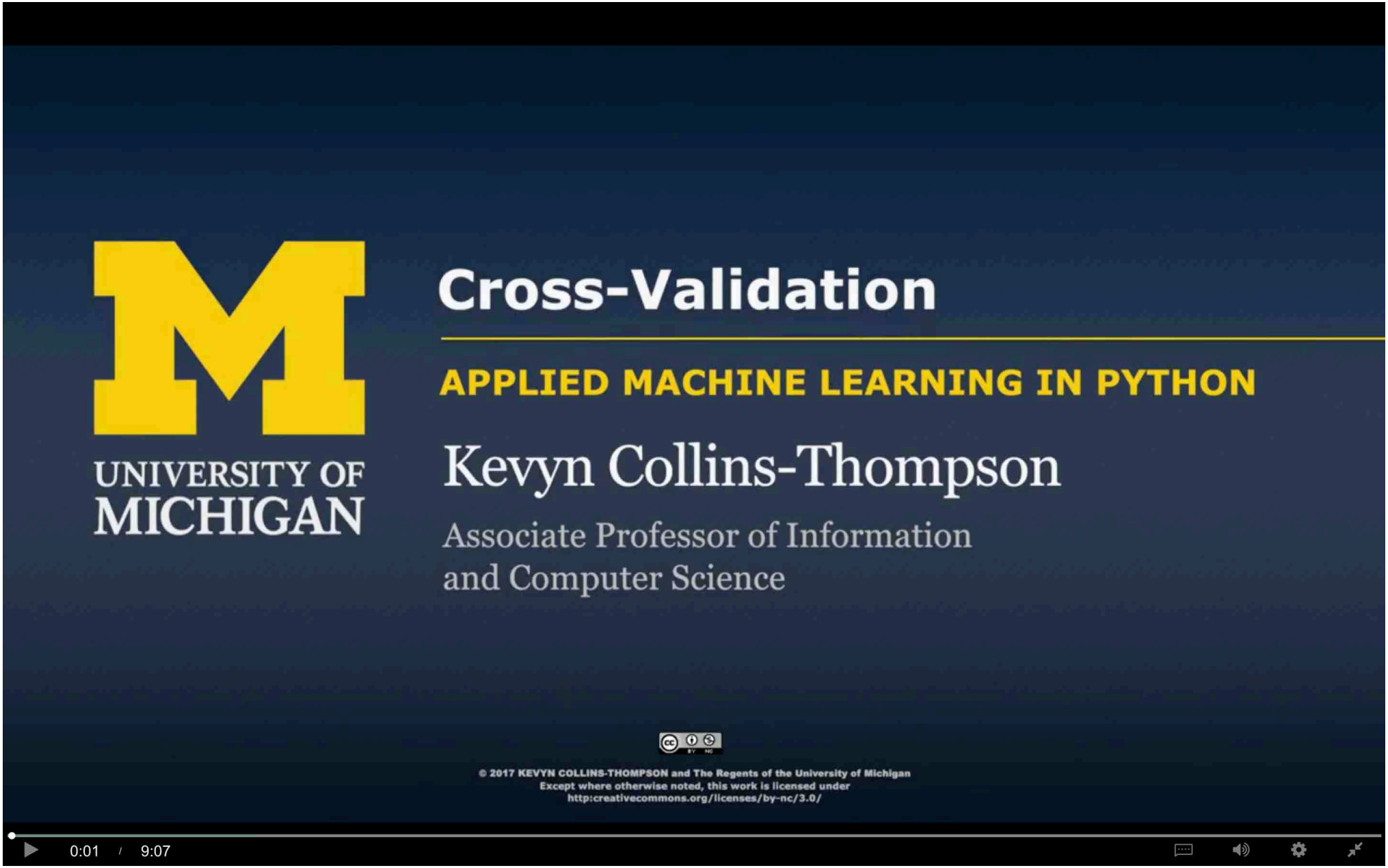
Cons:

- Efficiency (runtime speed and memory usage) decreases as training set size increases (e.g. over 50000 samples).
- Needs careful normalization of input data and parameter tuning.
- Does not provide direct probability estimates (but can be estimated using e.g. Platt scaling).
- Difficult to interpret why a prediction was made.

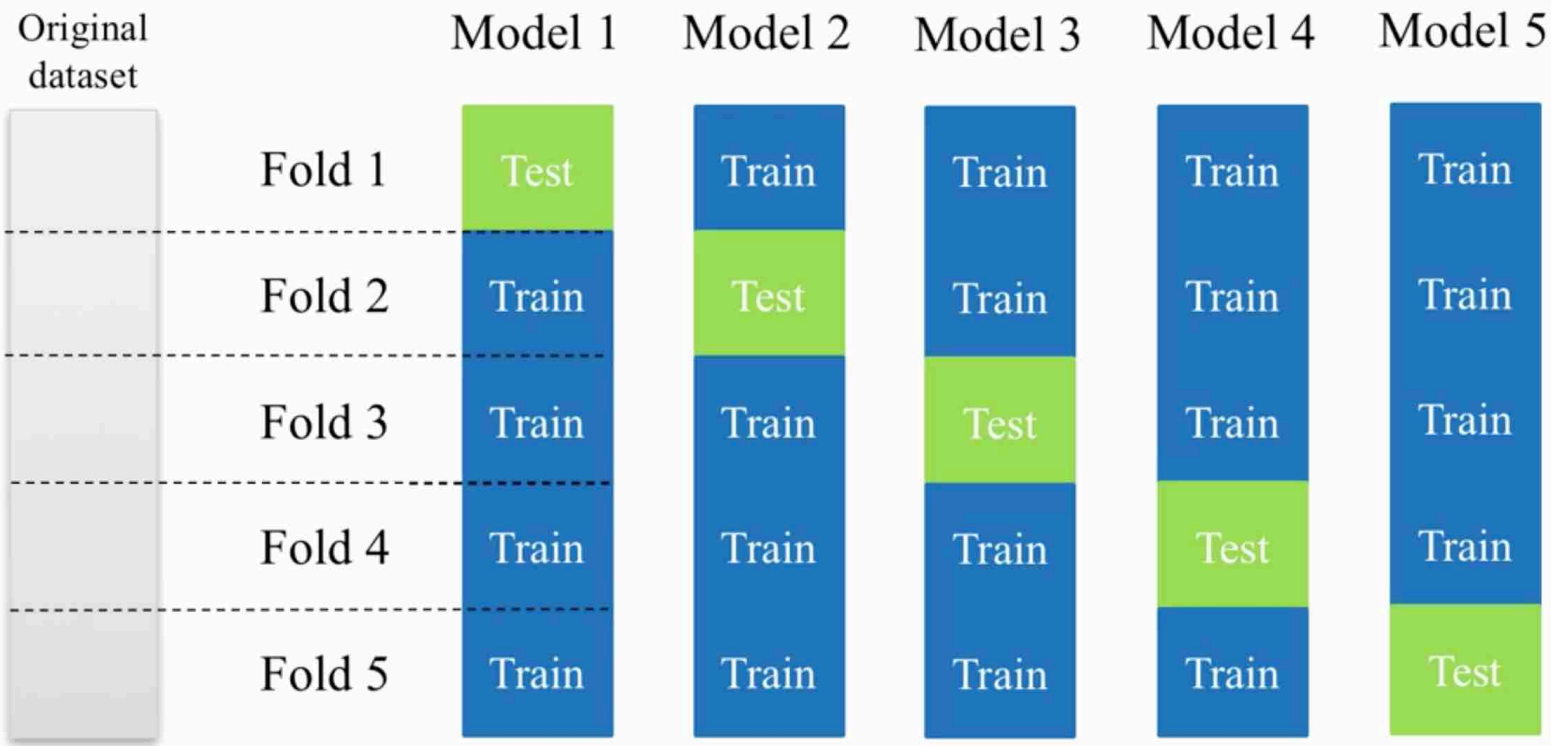
Kernelized Support Vector Machines (SVC): Important parameters

Model complexity

- `kernel`: Type of kernel function to be used
  - *Default = 'rbf' for radial basis function*
  - *Other types include 'polynomial'*
- kernel parameters
  - `gamma` *(γ): RBF kernel width*
- `C`: regularization parameter
- Typically `C` and `gamma` are tuned at the same time.


UNIVERSITY OF MICHIGAN
Cross-Validation
APPLIED MACHINE LEARNING IN PYTHON
Kevyn Collins-Thompson
Associate Professor of Information and Computer Science
© 2017 KEVYN COLLINS-THOMPSON and The Regents of the University of Michigan
Except where otherwise noted, this work is licensed under
http:creativecommons.org/licenses/by-nc/3.0/
0:01 / 9:07

Cross-validation Example (5-fold)



```
RBF-kernel SVC (with MinMax scaling) test set accuracy: 0.96
```

Cross-validation

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

In [30]:
```
from sklearn.model_selection import cross_val_score

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)

print('Cross-validation scores (3-fold):', cv_scores)
print('Mean cross-validation score (3-fold): {:.3f}'
     .format(np.mean(cv_scores)))
```

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

In [ ]:

Stratified Cross-validation

fruit_label	fruit_name
1	Apple
1	Apple
1	Apple
1	Apple
1	Apple
2	Mandarin
...	...
3	Orange
...	...
4	Lemon
4	Lemon
4	Lemon
4	Lemon
4	Lemon

(Folds and dataset shortened for illustration purposes.)

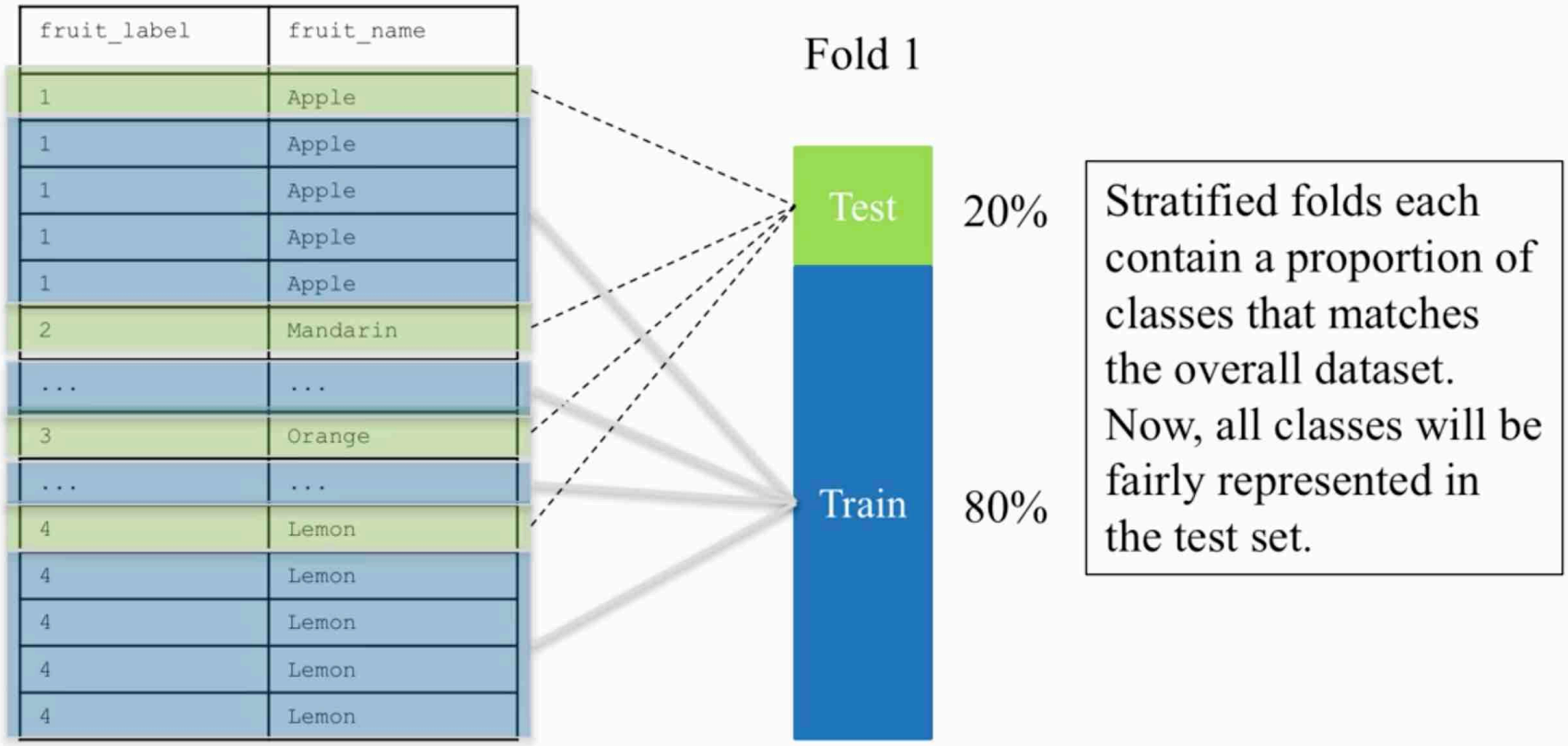
Example has 20 data samples
= 4 classes with 5 samples each.

5-fold CV: 5 folds of 4 samples each.

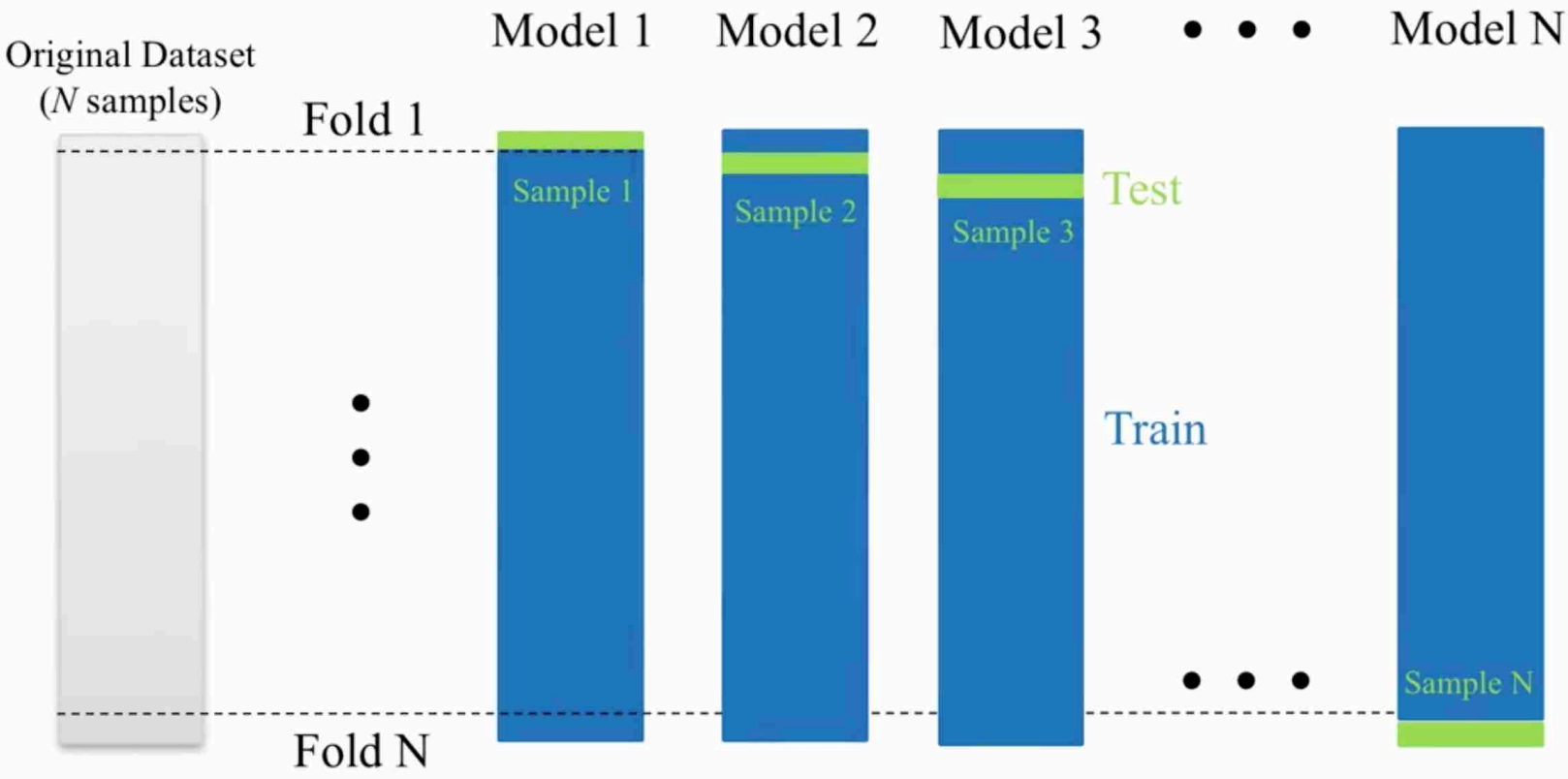
Fold 1 uses the first 20% of the dataset as the test set, which only contains samples from class 1.

Classes 2, 3, 4 are missing entirely from test set and so will be missing from the evaluation.

Stratified Cross-validation



Leave-one-out cross-validation (with *N* samples in dataset)



Validation curves show sensitivity to changes in an important parameter

```
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=5)
```

```
: print(train_scores)
```

```
[[ 0.48648649  0.425       0.41463415]
 [ 0.83783784  0.725       0.75609756]
 [ 0.91891892  0.9         0.92682927]
 [ 1.          1.          0.97560976]]
```

```
: print(test_scores)
```

```
[[ 0.45454545  0.31578947  0.33333333]
 [ 0.81818182  0.68421053  0.61111111]
 [ 0.40909091  0.84210526  0.66666667]
 [ 0.36363636  0.21052632  0.38888889]]
```

One row per parameter sweep value,
One column per CV fold.

```
cv_scores = cross_val_score(clf, X, y)

print('Cross-validation scores (3-fold):', cv_scores)
print('Mean cross-validation score (3-fold): {:.3f}'
      .format(np.mean(cv_scores)))
```

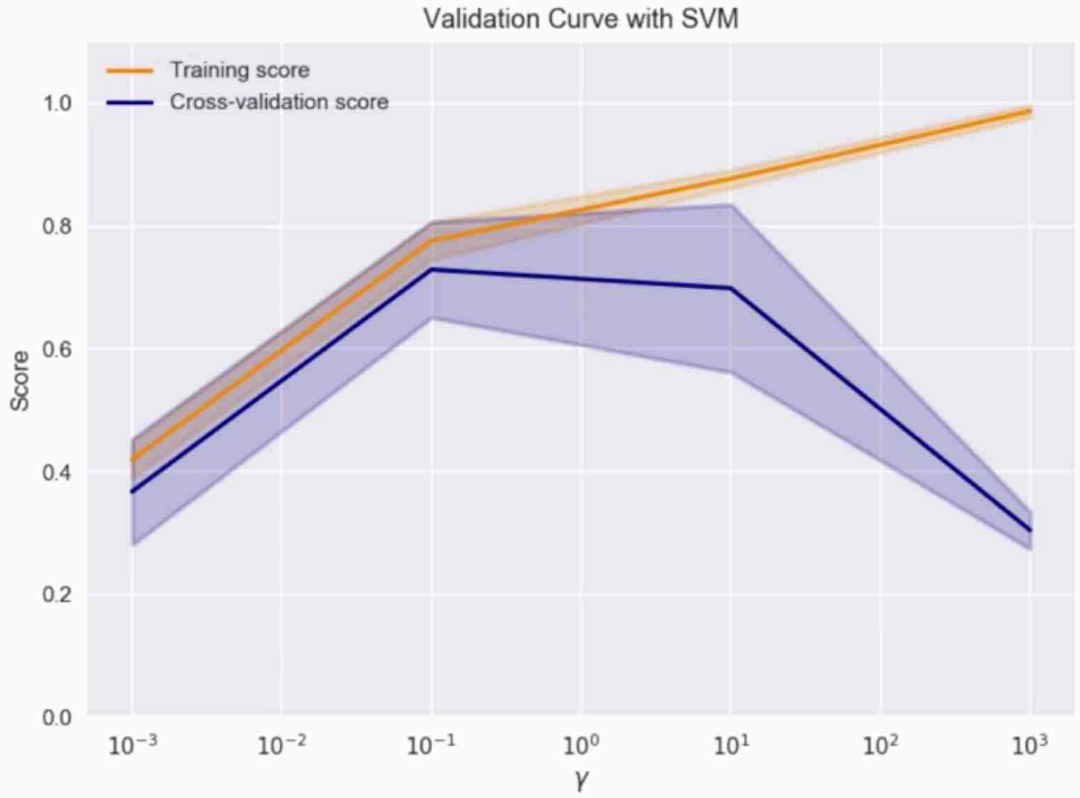
```
Cross-validation scores (3-fold): [ 0.77  0.74  0.83]
Mean cross-validation score (3-fold): 0.781
```

Validation curve example

In [31]:
```
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 [ ]:
```

```

Validation Curve Example



The validation curve shows the mean cross-validation accuracy (solid lines) for training (orange) and test (blue) sets as a function of the SVM parameter (gamma). It also shows the variation around the mean (shaded region) as computed from k-fold cross-validation scores.

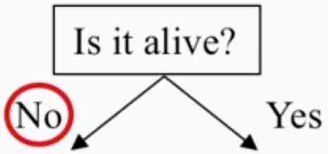
Decision Trees

APPLIED MACHINE LEARNING IN PYTHON

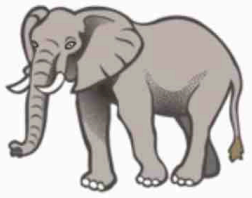
Kevyn Collins-Thompson

Associate Professor of Information
and Computer Science

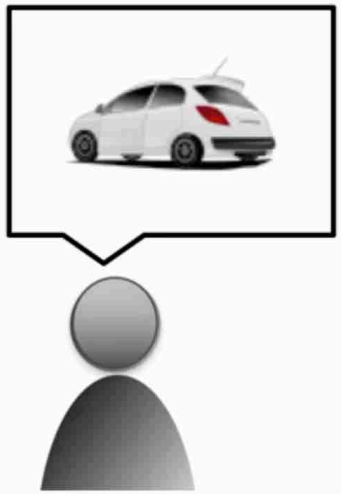
Decision Tree Example



Objects with Alive == No

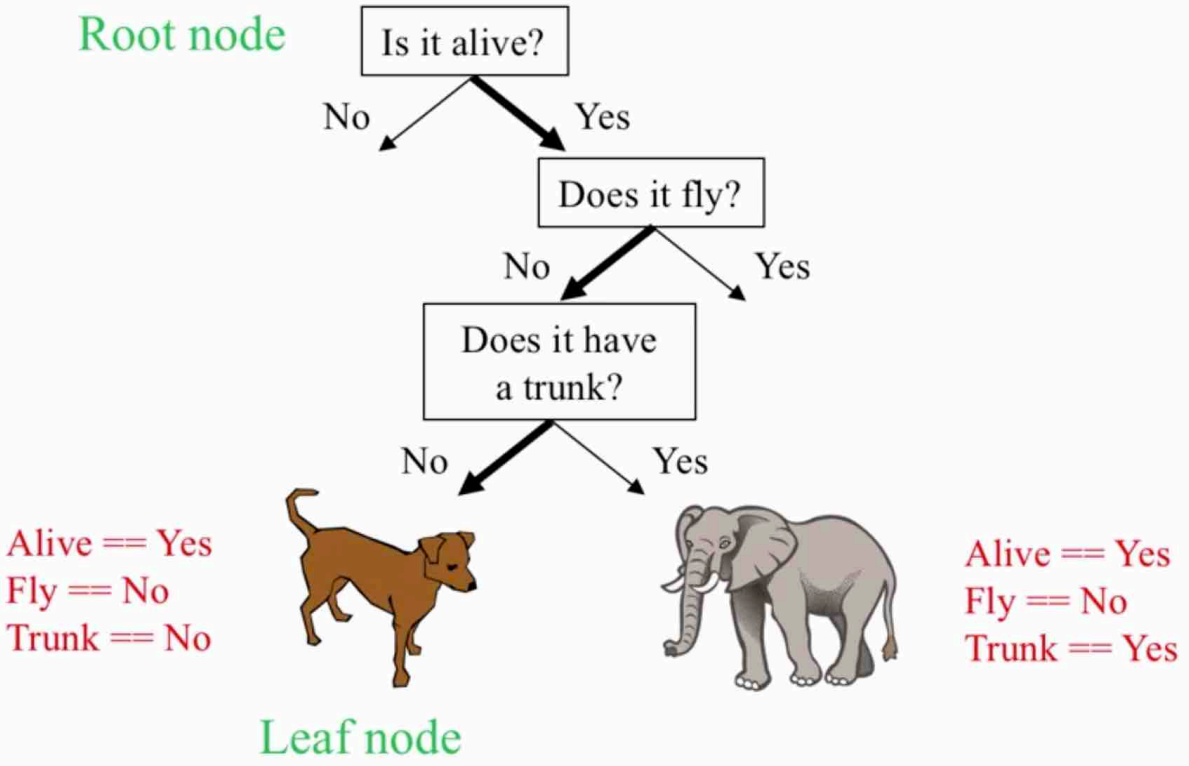


Decision Tree Example



Objects with: Alive == No AND Fly == No AND Carry > 10 == No

Decision Tree Example



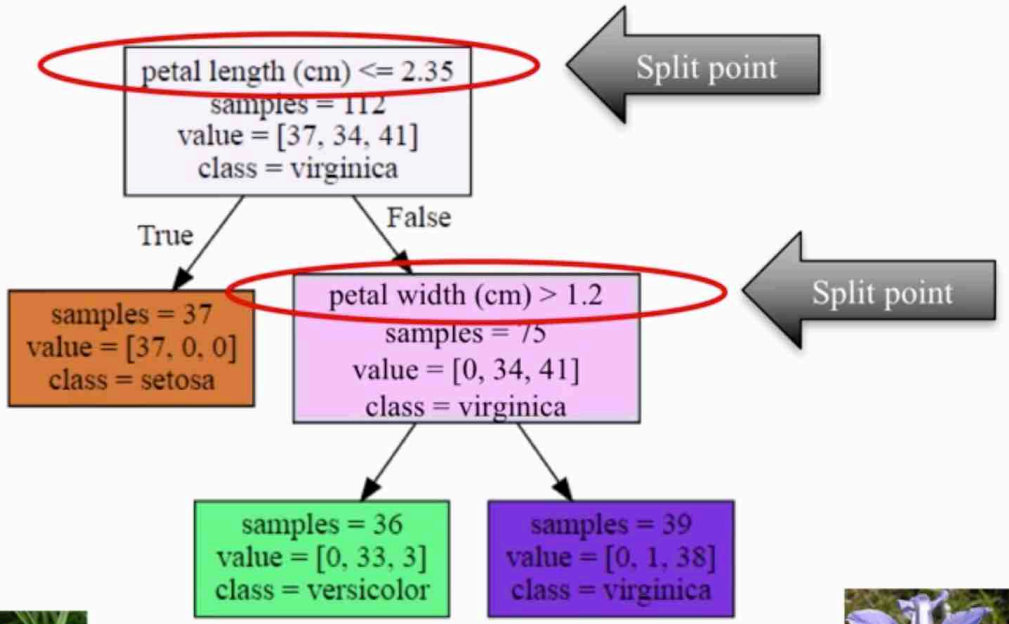
The Iris Dataset

150 flowers
3 species
50 examples/species

Iris setosa *Iris versicolor* *Iris virginica*

Photo credit: Radomił Binek via https://en.wikipedia.org/wiki/Iris_flower_data_set

Decision Tree Splits



samples at this leaf have:
petal length <= 2.35

setosa

versicolor

samples at this leaf have:
petal length > 2.35
AND petal width <= 1.2

samples at this leaf have:
petal length > 2.35
AND petal width > 1.2

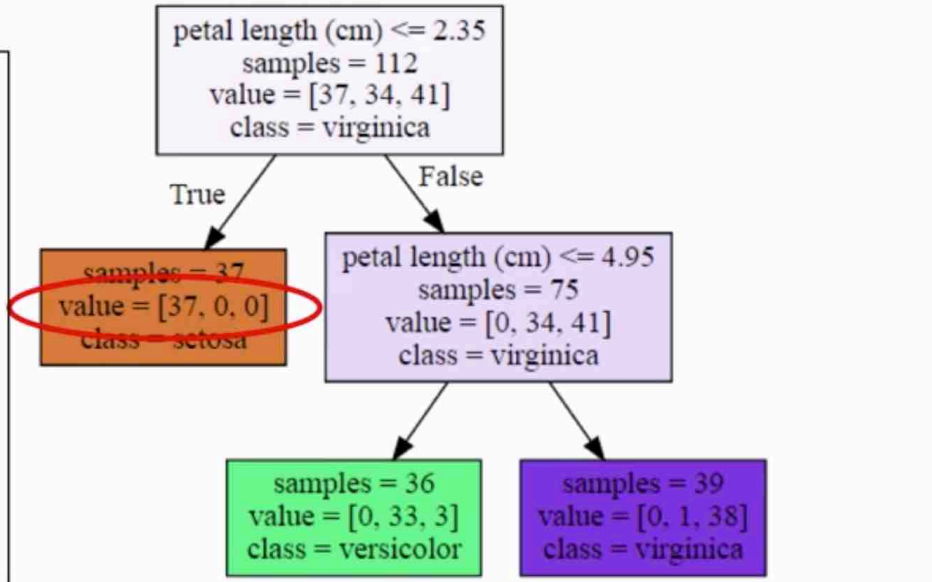
virginica

Informativeness of Splits

The *value* list gives the number of samples of each class that end up at this leaf node during training.

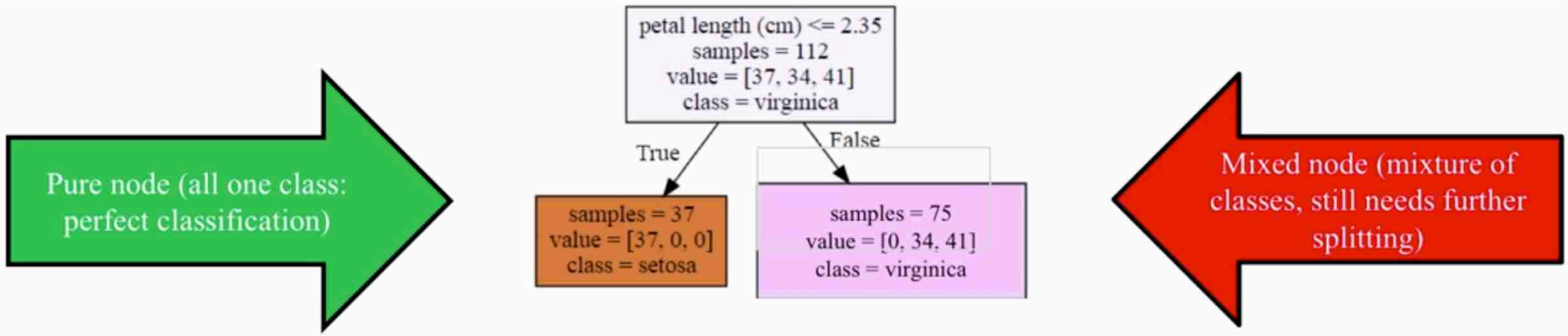
The iris dataset has 3 classes, so there are three counts.

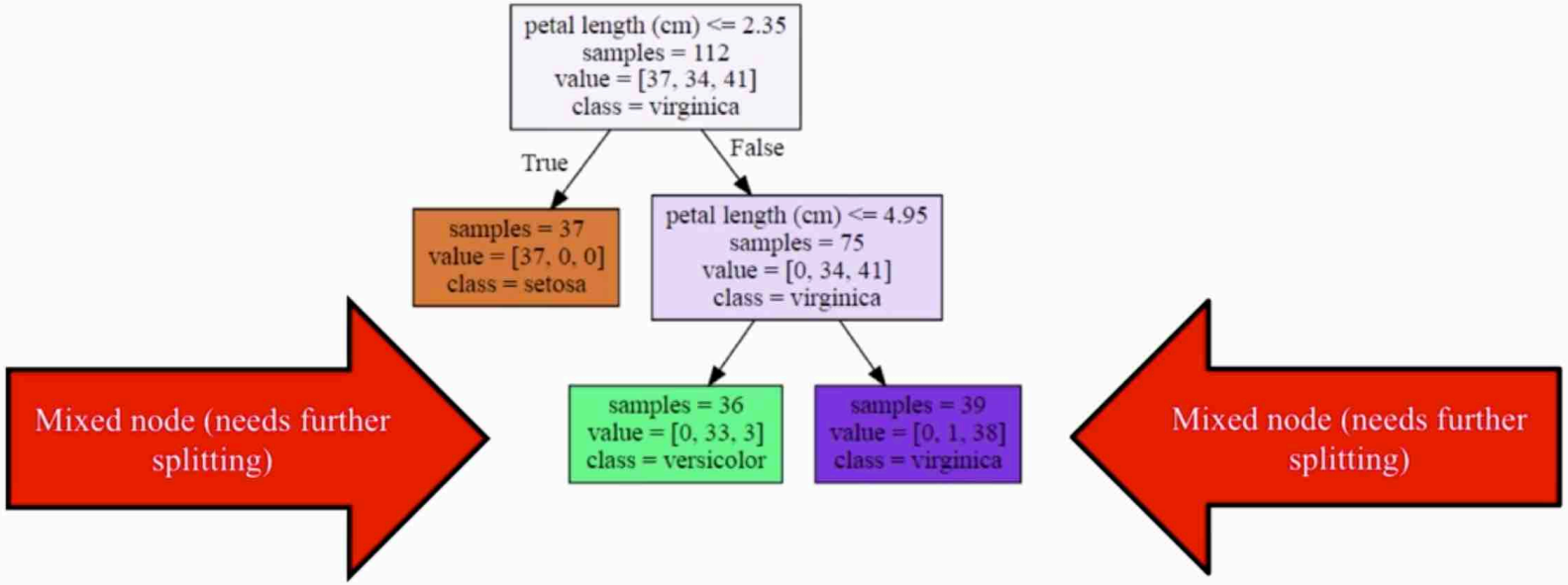
This leaf has 37 setosa samples, zero versicolor, and zero virginica samples.



This leaf has 0 setosa, 33 versicolor, and 3 virginica samples.

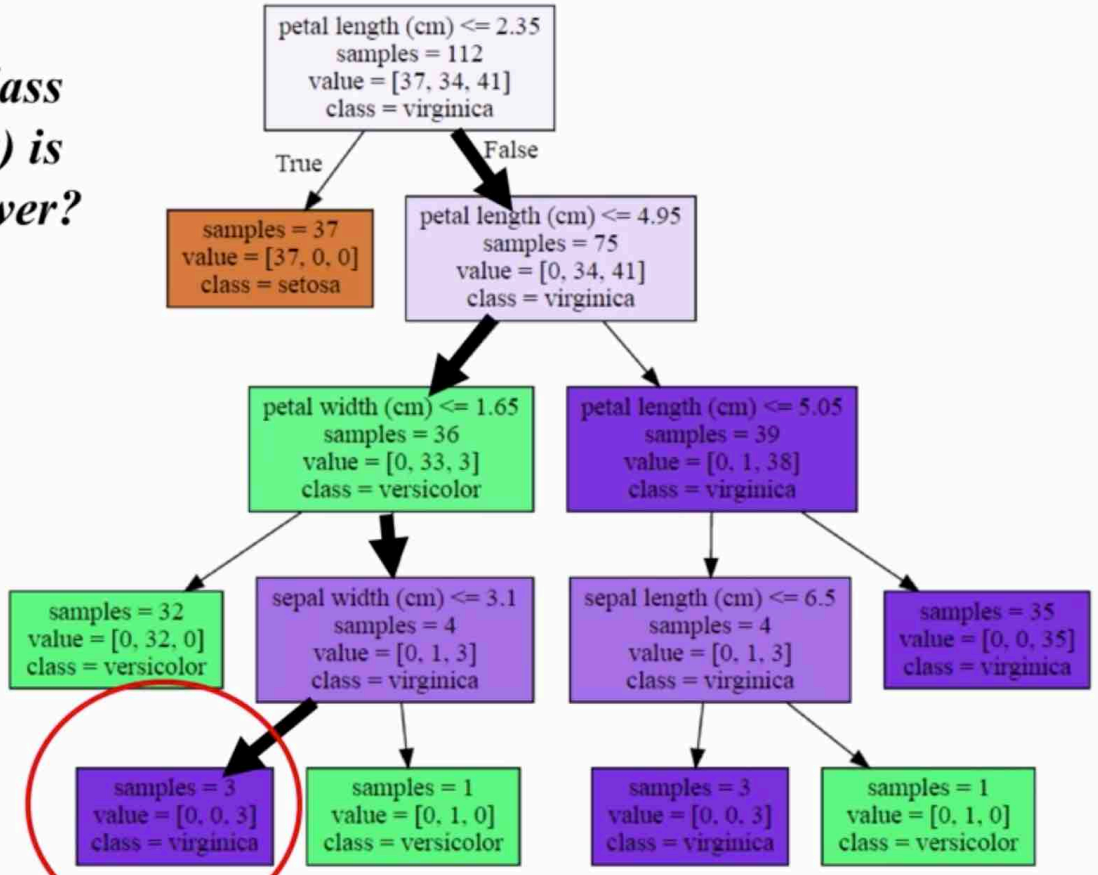
This leaf has 0 setosa, 1 versicolor, and 38 virginica samples.


Pure node (all one class: perfect classification)
petal length (cm) <= 2.35
samples = 112
value = [37, 34, 41]
class = virginica
True
False
samples = 37
value = [37, 0, 0]
class = setosa
samples = 75
value = [0, 34, 41]
class = virginica
Mixed node (mixture of classes, still needs further splitting)


petal length (cm) <= 2.35
samples = 112
value = [37, 34, 41]
class = virginica
True
False
samples = 37
value = [37, 0, 0]
class = setosa
petal length (cm) <= 4.95
samples = 75
value = [0, 34, 41]
class = virginica
samples = 36
value = [0, 33, 3]
class = versicolor
samples = 39
value = [0, 1, 38]
class = virginica
Mixed node (needs further splitting)
Mixed node (needs further splitting)

What class (species) is this flower?

petal length: 3.0
petal width: 2.0
sepal width: 2.0
sepal length: 4.2



Leaf counts are: setosa = 0, versicolor = 0, virginica = 3
Predicted class is majority class at this leaf: **virginica**

Decision Trees

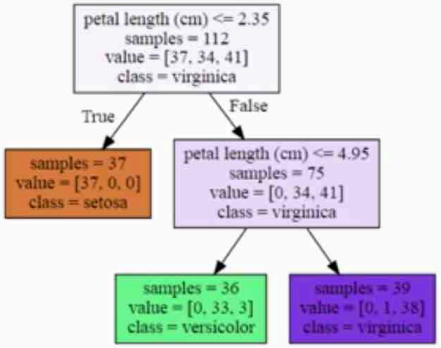
In [5]:
```
from sklearn.datasets import load_iris
from sklearn.tree import DecisionTreeClassifier
from adspy_shared_utilities import plot_decision_tree
from sklearn.model_selection import train_test_split


iris = load_iris()

X_train, X_test, y_train, y_test = train_test_split(iris.data, iris.target,
                                                    random_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: 0.97
```

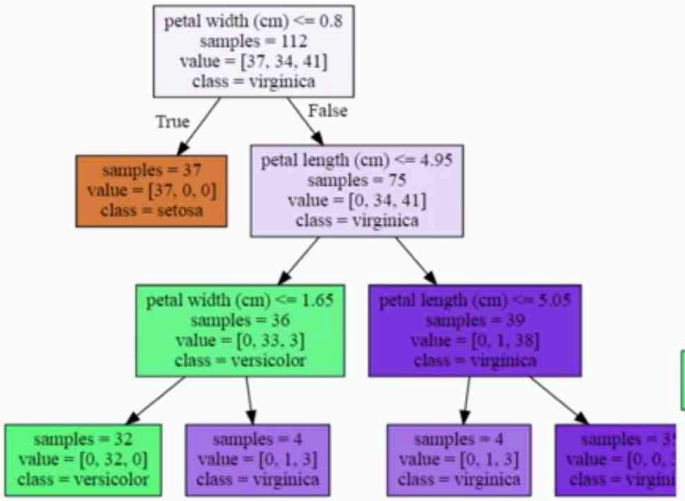
In [ ]:

Controlling the Model Complexity of Decision Trees

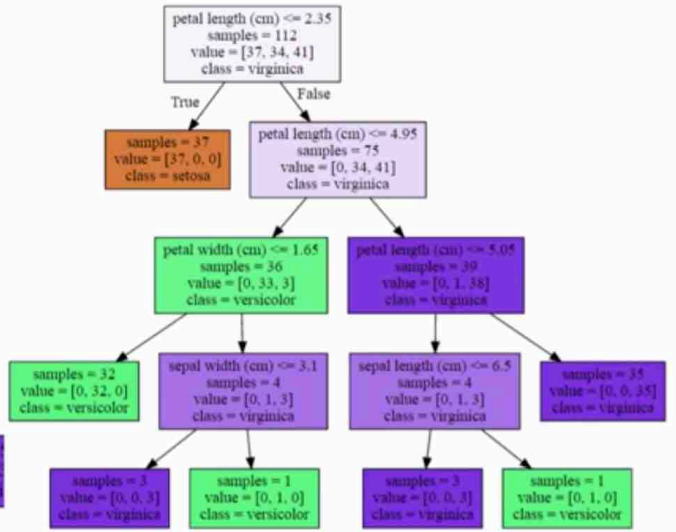
`max_depth = 2`



`max_depth = 3`



`max_depth = 4`



Other parameters: Max. # of leaf nodes: `max_leaf_nodes` Min. samples to consider splitting: `min_samples_leaf`

```
     .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: 0.97
```

Setting max decision tree depth to help avoid overfitting

In [6]:
```
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
```

In [ ]:

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

Setting max decision tree depth to help avoid overfitting

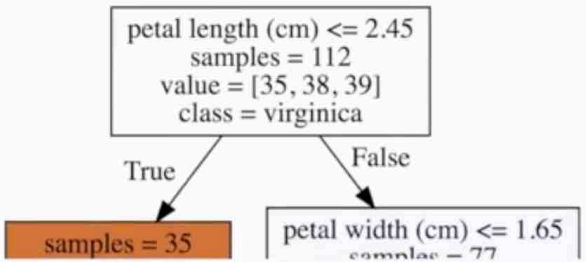
In [6]:
```
clf2 = DecisionTreeClassifier(max_depth = 3).fit(X_train, y_train)

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)))
```

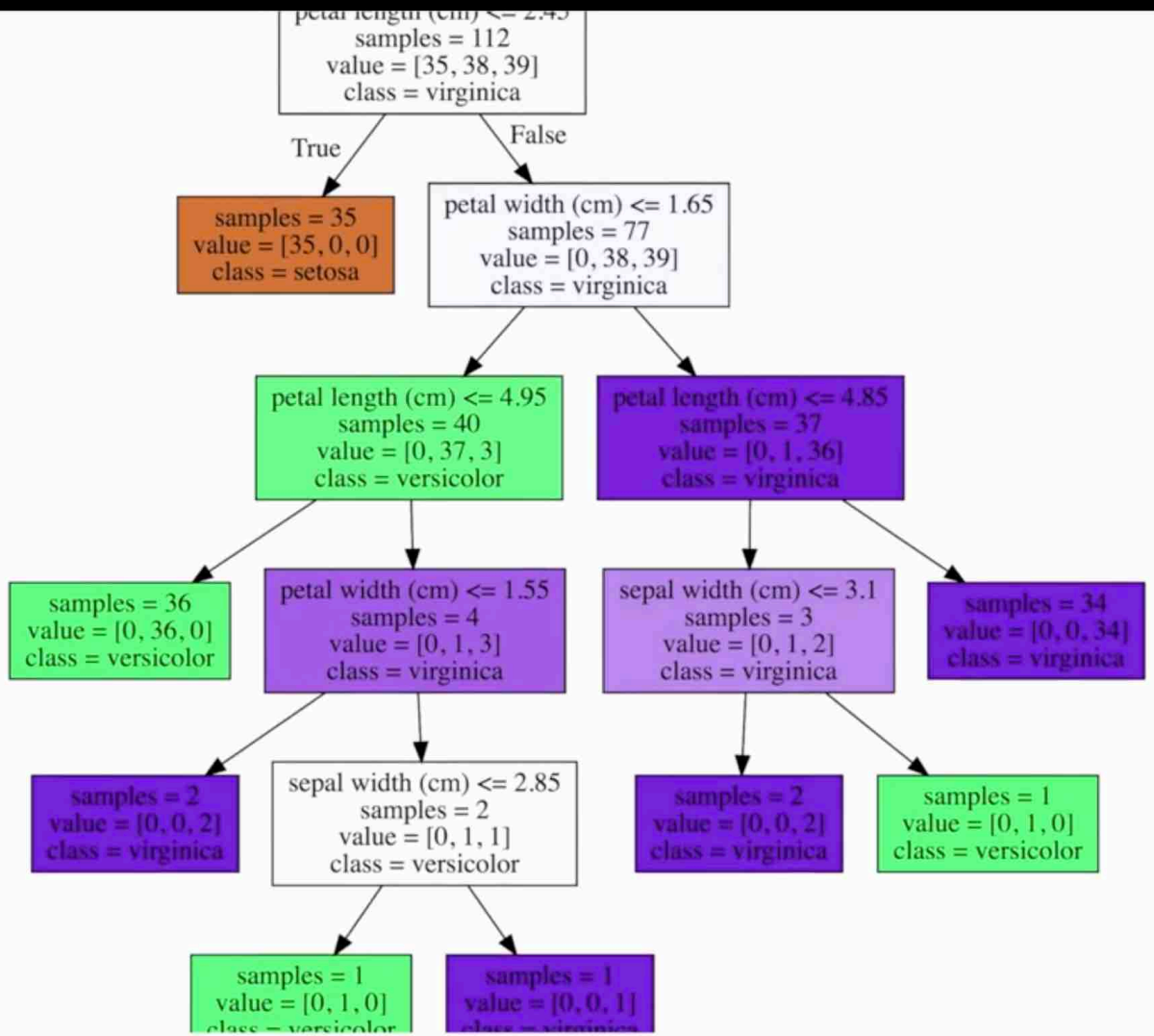
```
Accuracy of Decision Tree classifier on training set: 0.98
Accuracy of Decision Tree classifier on test set: 0.97
```

Visualizing decision trees

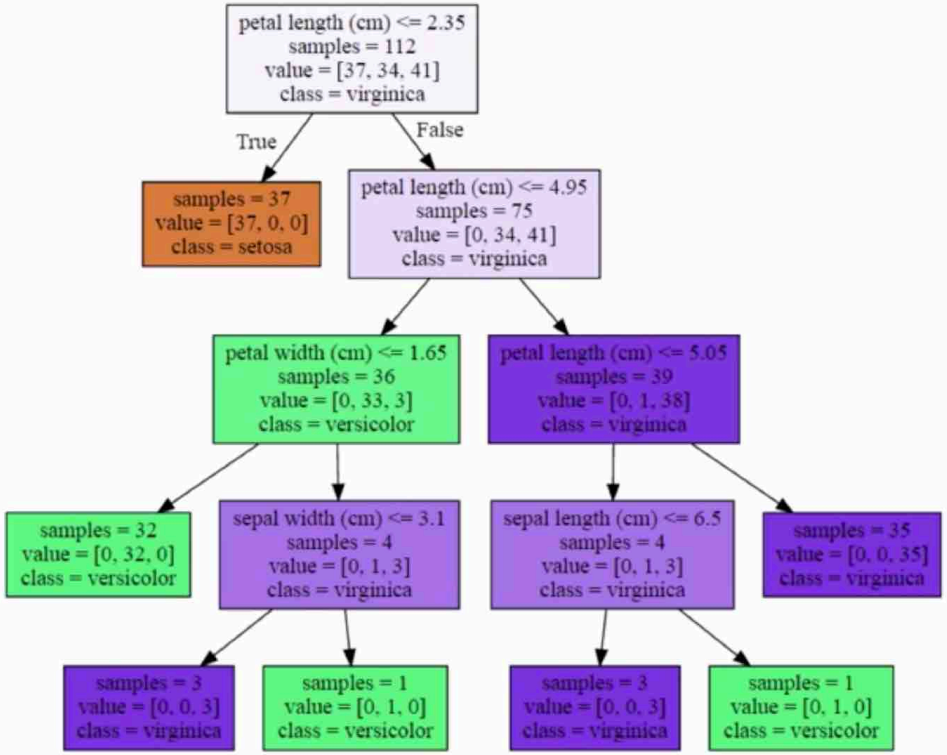
In [7]:
```
plot_decision_tree(clf, iris.feature_names, iris.target_names)
```

Out[7]:




petal length (cm) <= 2.45
samples = 112
value = [35, 38, 39]
class = virginica
True
False
samples = 35
value = [35, 0, 0]
class = setosa
petal width (cm) <= 1.65
samples = 77
value = [0, 38, 39]
class = virginica
petal length (cm) <= 4.95
samples = 40
value = [0, 37, 3]
class = versicolor
petal length (cm) <= 4.85
samples = 37
value = [0, 1, 36]
class = virginica
samples = 36
value = [0, 36, 0]
class = versicolor
petal width (cm) <= 1.55
samples = 4
value = [0, 1, 3]
class = virginica
sepal width (cm) <= 3.1
samples = 3
value = [0, 1, 2]
class = virginica
samples = 34
value = [0, 0, 34]
class = virginica
samples = 2
value = [0, 0, 2]
class = virginica
sepal width (cm) <= 2.85
samples = 2
value = [0, 1, 1]
class = versicolor
samples = 2
value = [0, 0, 2]
class = virginica
samples = 1
value = [0, 1, 0]
class = versicolor
samples = 1
value = [0, 1, 0]
samples = 1
value = [0, 0, 1]

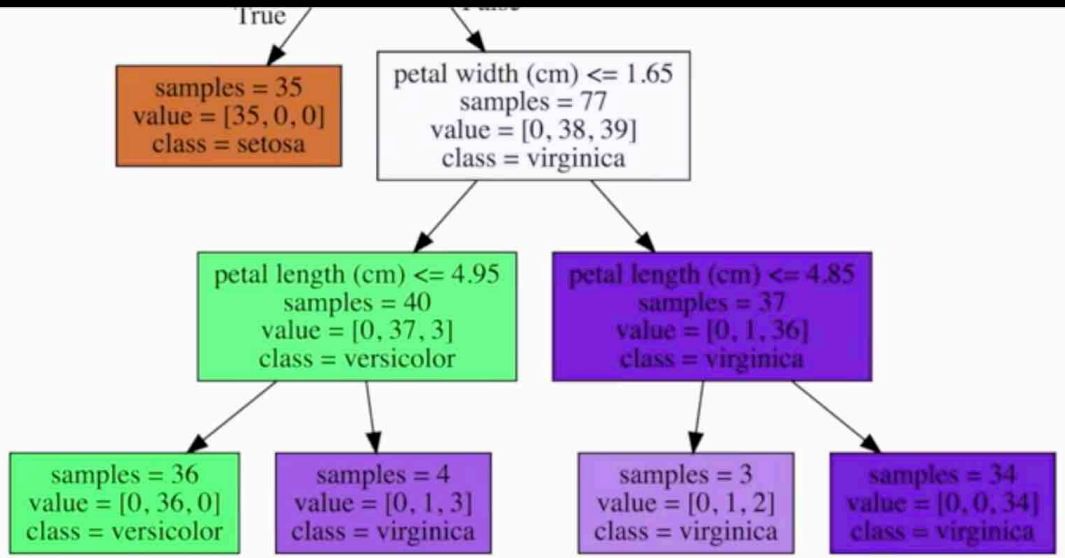
Visualizing Decision Trees



See: `plot_decision_tree()` function in `adspy_shared_utilities.py` code

Feature Importance: How important is a feature to overall prediction accuracy?

- A number between 0 and 1 assigned to each feature.
- Feature importance of 0 ⇨ the feature was not used in prediction.
- Feature importance of 1 ⇨ the feature predicts the target perfectly.
- All feature importances are normalized to sum to 1.



Feature importance

In [ ]:

```
from adspy_shared_utilities import plot_feature_importances

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

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

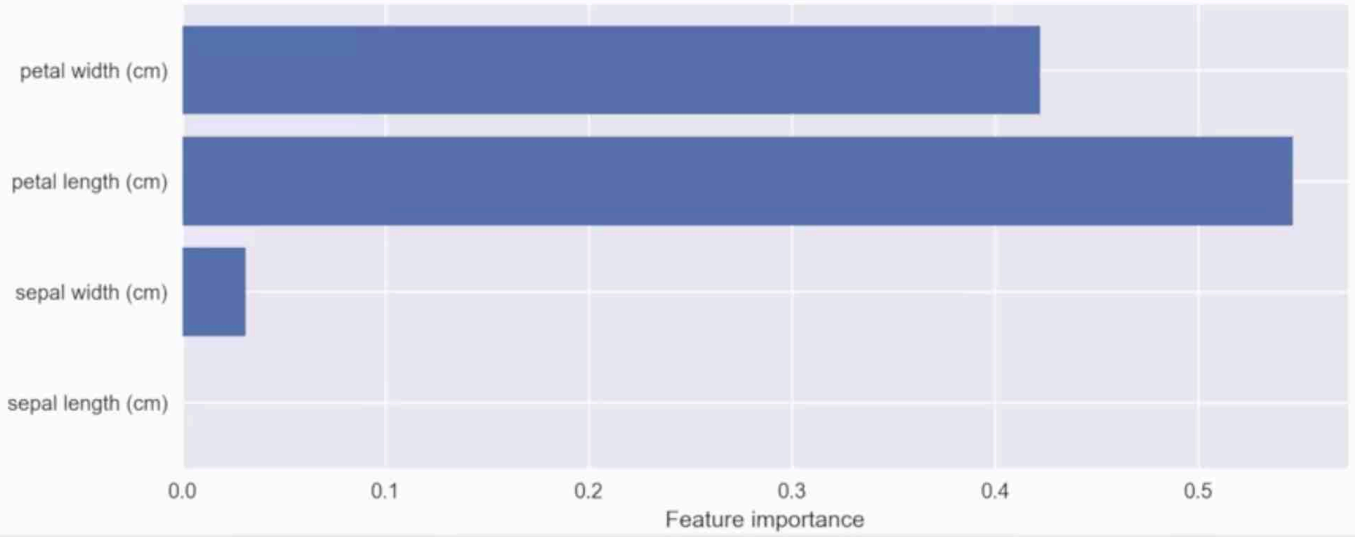
15:06

15:06 / 19:40

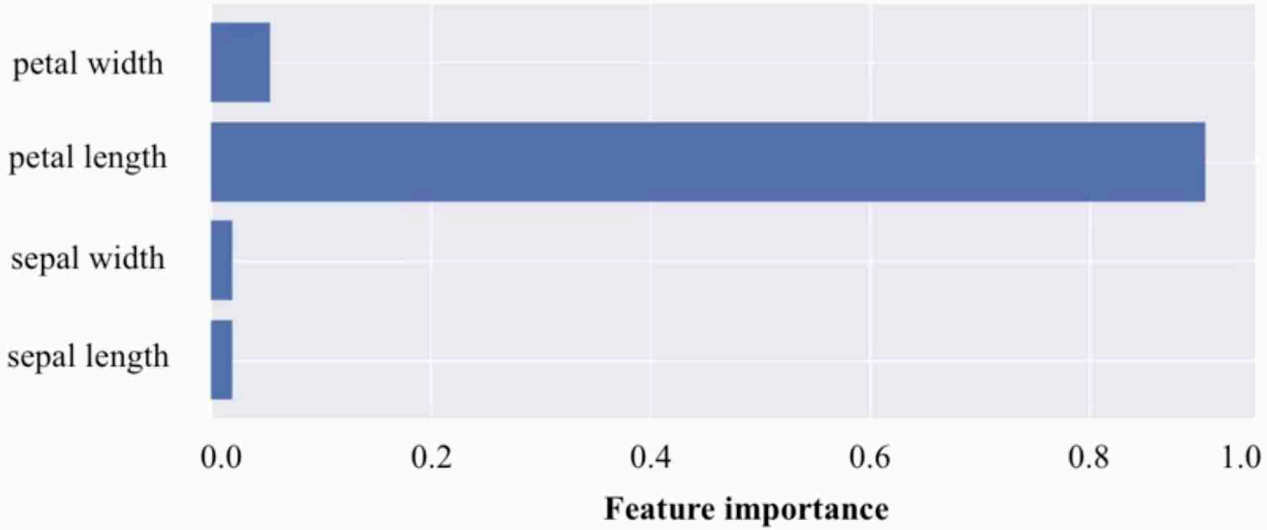
In [9]:

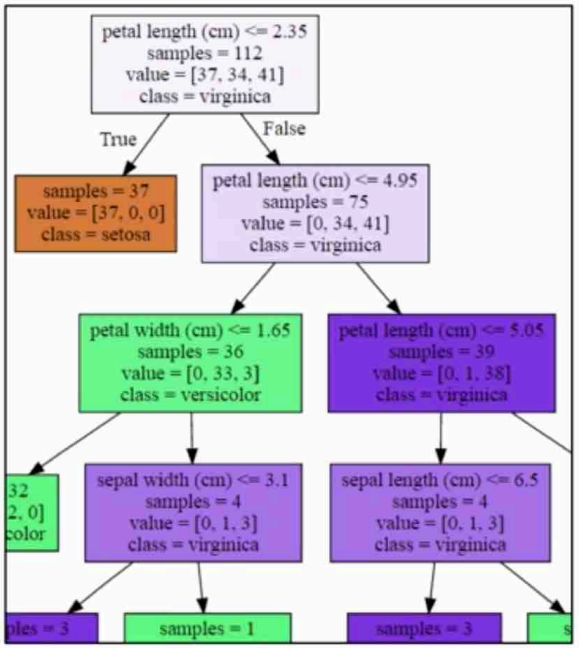
```
from adspy_shared_utilities import plot_feature_importances

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

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

Figure 7



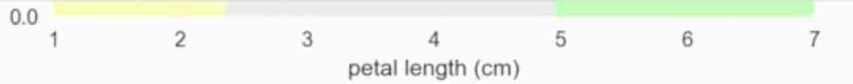
Feature Importance Chart





Decision tree

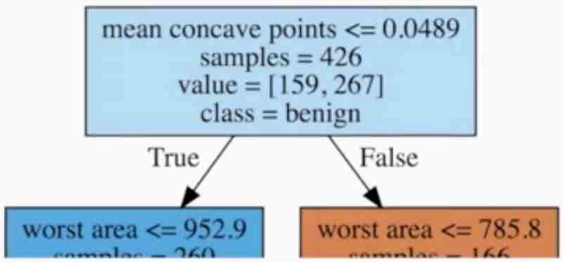
See: `plot_feature_importances()` function in `adspy_shared_utilities.py` code



Decision Trees on a real-world dataset

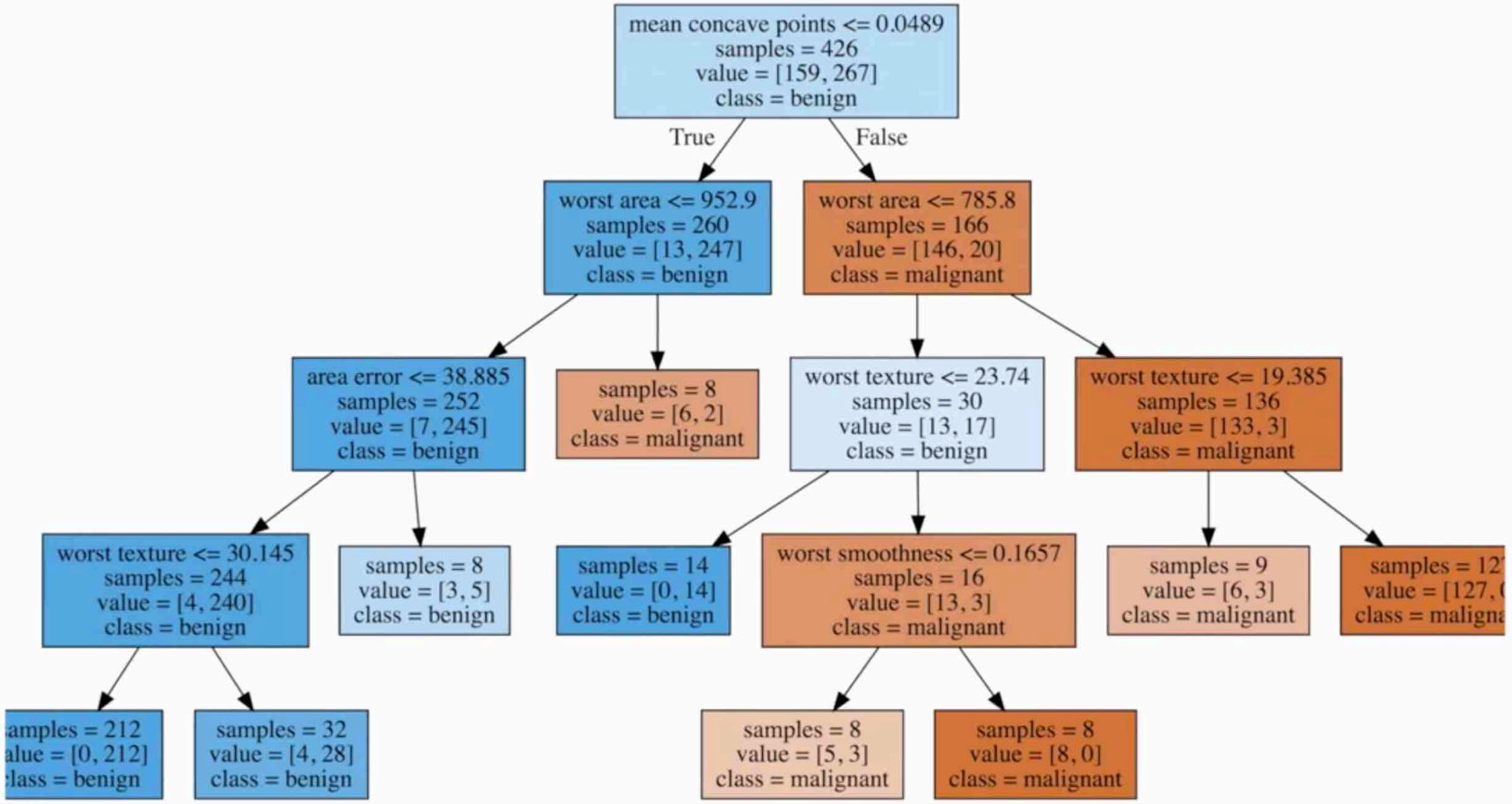
In [11]:
```
from sklearn.tree import DecisionTreeClassifier
from adspy_shared_utilities import plot_decision_tree
from adspy_shared_utilities import plot_feature_importances

X_train, X_test, y_train, y_test = train_test_split(X_cancer, y_cancer,
                                                    random_state = 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[11]:



16:46 / 19:40

Out[11]:



In [ ]:

17:24

17:24 / 19:40

Decision Trees: Pros and Cons

Pros:

- Easily visualized and interpreted.
- No feature normalization or scaling typically needed.
- Work well with datasets using a mixture of feature types (continuous, categorical, binary)

Cons:

- Even after tuning, decision trees can often still overfit.
- Usually need an ensemble of trees for better generalization performance.

Decision Trees: DecisionTreeClassifier Key Parameters

- `max_depth`: controls maximum depth (number of split points). Most common way to reduce tree complexity and overfitting.
- `min_samples_leaf`: threshold for the minimum # of data instances a leaf can have to avoid further splitting.
- `max_leaf_nodes`: limits total number of leaves in the tree.
- In practice, adjusting only one of these (e.g. `max_depth`) is enough to reduce overfitting.