Agenda

- How do I choose **which model to use** for my supervised learning task?
- How do I choose the **best tuning parameters** for that model?
- How do I estimate the **likely performance of my model** on out-of-sample data?

Review

- Classification task: Predicting the species of an unknown iris
- Used three classification models: KNN (K=1), KNN (K=5), logistic regression
- Need a way to choose between the models

Solution: Model evaluation procedures

Evaluation procedure #1: Train and test on the entire dataset

1. Train the model on the **entire dataset**.
2. Test the model on the **same dataset**, and evaluate how well we did by comparing the **predicted** response values with the **true** response values.

In [ ]:

```python
import numpy as np
# read in the iris data
from sklearn.datasets import load_iris
iris = load_iris()

# create X (features) and y (response)
X = iris.data
y = iris.target
```

Logistic regression

In [ ]:

```python
# import the class
from sklearn.linear_model import LogisticRegression

# instantiate the model (using the default parameters)
logreg = LogisticRegression()

# fit the model with data
logreg.fit(X, y)

# predict the response values for the observations in X
logreg.predict(X)
```

In [ ]:

```python
# store the predicted response values
y_pred = logreg.predict(X)

# check how many predictions were generated
len(y_pred)
```

Classification accuracy:

- **Proportion** of correct predictions
- Common **evaluation metric** for classification problems

In [ ]:

```python
# compute classification accuracy for the logistic regression model
from sklearn import metrics
print metrics.accuracy_score(y, y_pred)
```

- Known as **training accuracy** when you train and test the model on the same data

KNN (K=5)

In [ ]:

```
from sklearn.neighbors import KNeighborsClassifier
knn = KNeighborsClassifier(n_neighbors=5)
knn.fit(X, y)
y_pred = knn.predict(X)
print metrics.accuracy_score(y, y_pred)
```

KNN (K=1)

In [ ]:

```
knn = KNeighborsClassifier(n_neighbors=1)
knn.fit(X, y)
y_pred = knn.predict(X)
print metrics.accuracy_score(y, y_pred)
```

Problems with training and testing on the same data

- Goal is to estimate likely performance of a model on **out-of-sample data**
- But, maximizing training accuracy rewards **overly complex models** that won't necessarily generalize
- Unnecessarily complex models **overfit** the training data

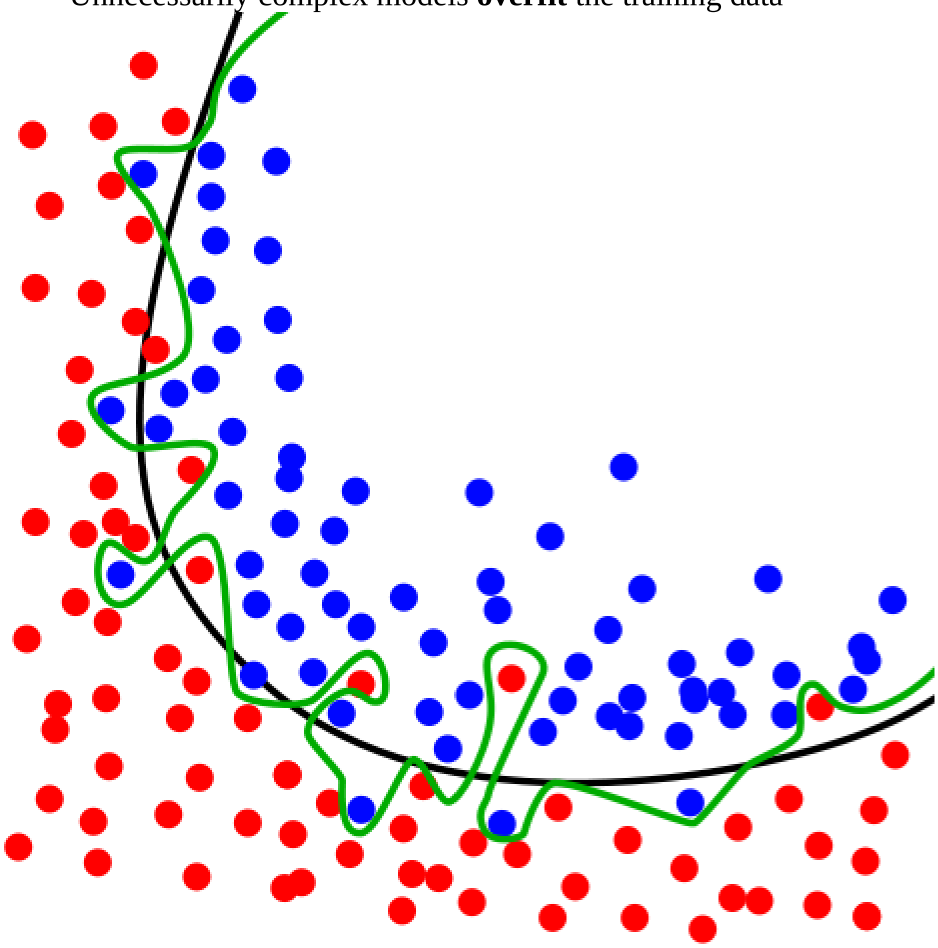

Image Credit: Overfitting by Chabacano. Licensed under GFDL via Wikimedia Commons.

Evaluation procedure #2: Train/test split

1. Split the dataset into two pieces: a **training set** and a **testing set**.

2. Train the model on the **training set**.
3. Test the model on the **testing set**, and evaluate how well we did.

In [ ]:

```
# print the shapes of X and y
print X.shape
print y.shape
```

In [ ]:

```
# STEP 1: split X and y into training and testing sets
from sklearn.cross_validation import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25,
random_state=4)
```

X_train
X_test

feature 1	feature 2	response
1	2	2
3	4	12
5	6	30
7	8	56
9	10	90

y_train
y_test

What did this accomplish?

- Model can be trained and tested on **different data**
- Response values are known for the testing set, and thus **predictions can be evaluated**
- **Testing accuracy** is a better estimate than training accuracy of out-of-sample performance

In [ ]:

```
# print the shapes of the new X objects
print X_train.shape
print X_test.shape
```

In [ ]:

```
# print the shapes of the new y objects
print y_train.shape
print y_test.shape
```

In [ ]:

```
# STEP 2: train the model on the training set
logreg = LogisticRegression()
logreg.fit(X_train, y_train)
```

In [ ]:

```
# STEP 3: make predictions on the testing set
y_pred = logreg.predict(X_test)

# compare actual response values (y_test) with predicted response values (y_pred)
print metrics.accuracy_score(y_test, y_pred)
```

Repeat for KNN with K=5:

In [ ]:

```
knn = KNeighborsClassifier(n_neighbors=5)
knn.fit(X_train, y_train)
y_pred = knn.predict(X_test)
print metrics.accuracy_score(y_test, y_pred)
```

Repeat for KNN with K=1:

In [ ]:

```
knn = KNeighborsClassifier(n_neighbors=1)
knn.fit(X_train, y_train)
y_pred = knn.predict(X_test)
print metrics.accuracy_score(y_test, y_pred)
```

Can we locate an even better value for K?

In [ ]:

```
# try K=1 through K=25 and record testing accuracy
k_range = range(1, 26)
scores = []
for k in k_range:
    knn = KNeighborsClassifier(n_neighbors=k)
    knn.fit(X_train, y_train)
    y_pred = knn.predict(X_test)
    scores.append(metrics.accuracy_score(y_test, y_pred))
```

In [ ]:

```
# import Matplotlib (scientific plotting library)
import matplotlib.pyplot as plt

# allow plots to appear within the notebook
%matplotlib inline
```

In [ ]:

```
# plot the relationship between K and testing accuracy
fig, ax = plt.subplots()
ax.plot(k_range, scores)
ax.set_xlabel('Value of K for KNN')
ax.set_ylabel('Testing Accuracy')
ax.grid();
```

In [ ]:

```
scores
```

- **Training accuracy** rises as model complexity increases
- **Testing accuracy** penalizes models that are too complex or not complex enough
- For KNN models, complexity is determined by the **value of K** (lower value = more complex)

Making predictions on out-of-sample data

In [ ]:

```
knn.predict(np.array([3, 5, 4, 2]).reshape(1,4))
```

In [ ]:

```
# instantiate the model with the best known parameters
knn = KNeighborsClassifier(n_neighbors=11)

# train the model with X and y (not X_train and y_train)
knn.fit(X, y)

# make a prediction for an out-of-sample observation
knn.predict(np.array([3, 5, 4, 2]).reshape(1,4))
```

Downsides of train/test split?

- Provides a **high-variance estimate** of out-of-sample accuracy

- **K-fold cross-validation** overcomes this limitation
- But, train/test split is still useful because of its **flexibility and speed**