K-Nearest Neighbors with `scikit-learn`

Authors: Alex Sherman (DC)

Learning Objectives

1. Utilize the KNN model on the iris data set.
2. Implement scikit-learn's KNN model.
3. Assess the fit of a KNN Model using scikit-learn.

Lesson Guide

In this lesson, we will get an intuitive and practical feel for the **k-Nearest Neighbors** model. kNN is a **non-parametric model**. So, the model is not represented as an equation with parameters (e.g. the β values in linear regression).

First, we will make a model by hand to classify iris flower data. Next, we will automatedly make a model using kNN.

> You may have heard of the clustering algorithm **k-Means Clustering**. These techniques have nothing in common, aside from both having a parameter k!

Overview of the Iris Data Set

```
In [79]: # Read the iris data into a DataFrame.
         import pandas as pd
         import numpy as np

         data = 'iris.data'
         iris = pd.read_csv(data)
```

```
In [80]: iris.head()
```

Out[80]:

	sepal_length	sepal_width	petal_length	petal_width	species
0	5.1	3.5	1.4	0.2	Iris-setosa
1	4.9	3.0	1.4	0.2	Iris-setosa
2	4.7	3.2	1.3	0.2	Iris-setosa
3	4.6	3.1	1.5	0.2	Iris-setosa
4	5.0	3.6	1.4	0.2	Iris-setosa

Terminology

- **150 observations** (n=150): Each observation is one iris flower.
- **Four features** (p=4): sepal length, sepal width, petal length, and petal width.
- **Response**: iris species.
- **Classification problem** because response is categorical.

Exercise: "Human Learning" With Iris Data

Question: Can you predict the species of an iris using petal and sepal measurements?

1. Read the iris data into a Pandas DataFrame, including column names.
2. Gather some basic information about the data.
3. Use sorting, split-apply-combine, and/or visualization to look for differences between species.
4. Write down a set of rules that could be used to predict species based on iris measurements.

BONUS: Define a function that accepts a row of data and returns a predicted species. Then, use that function to make predictions for all existing rows of data and check the accuracy of your predictions.

```
In [81]: import pandas as pd
         import matplotlib.pyplot as plt

         # Display plots in the notebook.
         %matplotlib inline

         # Increase default figure and font sizes for easier viewing.
         plt.rcParams['figure.figsize'] = (8, 6)
         plt.rcParams['font.size'] = 14
```

Read the iris data into a pandas DataFrame, including column names.

In [82]:
```
# Define the URL from which to retrieve the data (as a string).
path = 'iris.data'

# Retrieve the CSV file and add the column names.
iris = pd.read_csv(path)
```

Gather some basic information about the data.

In [83]:
```
# Observe first five rows of data.
iris.head(30)
```

Out[83]:

	sepal_length	sepal_width	petal_length	petal_width	species
0	5.1	3.5	1.4	0.2	Iris-setosa
1	4.9	3.0	1.4	0.2	Iris-setosa
2	4.7	3.2	1.3	0.2	Iris-setosa
3	4.6	3.1	1.5	0.2	Iris-setosa
4	5.0	3.6	1.4	0.2	Iris-setosa
5	5.4	3.9	1.7	0.4	Iris-setosa
6	4.6	3.4	1.4	0.3	Iris-setosa
7	5.0	3.4	1.5	0.2	Iris-setosa
8	4.4	2.9	1.4	0.2	Iris-setosa
9	4.9	3.1	1.5	0.1	Iris-setosa
10	5.4	3.7	1.5	0.2	Iris-setosa
11	4.8	3.4	1.6	0.2	Iris-setosa
12	4.8	3.0	1.4	0.1	Iris-setosa
13	4.3	3.0	1.1	0.1	Iris-setosa
14	5.8	4.0	1.2	0.2	Iris-setosa
15	5.7	4.4	1.5	0.4	Iris-setosa
16	5.4	3.9	1.3	0.4	Iris-setosa
17	5.1	3.5	1.4	0.3	Iris-setosa
18	5.7	3.8	1.7	0.3	Iris-setosa
19	5.1	3.8	1.5	0.3	Iris-setosa
20	5.4	3.4	1.7	0.2	Iris-setosa
21	5.1	3.7	1.5	0.4	Iris-setosa
22	4.6	3.6	1.0	0.2	Iris-setosa
23	5.1	3.3	1.7	0.5	Iris-setosa
24	4.8	3.4	1.9	0.2	Iris-setosa
25	5.0	3.0	1.6	0.2	Iris-setosa
26	5.0	3.4	1.6	0.4	Iris-setosa
27	5.2	3.5	1.5	0.2	Iris-setosa
28	5.2	3.4	1.4	0.2	Iris-setosa
29	4.7	3.2	1.6	0.2	Iris-setosa

In [84]:
```
iris.shape
```

Out[84]:
```
(150, 5)
```

```
In [85]: iris.dtypes
```

```
Out[85]: sepal_length    float64
         sepal_width     float64
         petal_length    float64
         petal_width     float64
         species          object
         dtype: object
```

```
In [86]: iris.describe()
```

Out[86]:

	sepal_length	sepal_width	petal_length	petal_width
count	150.000000	150.000000	150.000000	150.000000
mean	5.843333	3.054000	3.758667	1.198667
std	0.828066	0.433594	1.764420	0.763161
min	4.300000	2.000000	1.000000	0.100000
25%	5.100000	2.800000	1.600000	0.300000
50%	5.800000	3.000000	4.350000	1.300000
75%	6.400000	3.300000	5.100000	1.800000
max	7.900000	4.400000	6.900000	2.500000

```
In [87]: iris.species.value_counts()
```

```
Out[87]: Iris-virginica     50
         Iris-setosa        50
         Iris-versicolor    50
         Name: species, dtype: int64
```

```
In [88]: iris.isnull().sum()
```

```
Out[88]: sepal_length    0
         sepal_width     0
         petal_length    0
         petal_width     0
         species         0
         dtype: int64
```

Use sorting, split-apply-combine, and/or visualization to look for differences between species.

```
In [89]: iris.head()
```

Out[89]:

	sepal_length	sepal_width	petal_length	petal_width	species
0	5.1	3.5	1.4	0.2	Iris-setosa
1	4.9	3.0	1.4	0.2	Iris-setosa
2	4.7	3.2	1.3	0.2	Iris-setosa
3	4.6	3.1	1.5	0.2	Iris-setosa
4	5.0	3.6	1.4	0.2	Iris-setosa

In [90]:
```
# Sort the DataFrame by petal_width.
iris.sort_values(by='petal_width', ascending=True, inplace=True)
```

In [91]:
```
iris.head()
```

Out[91]:

	sepal_length	sepal_width	petal_length	petal_width	species
32	5.2	4.1	1.5	0.1	Iris-setosa
13	4.3	3.0	1.1	0.1	Iris-setosa
37	4.9	3.1	1.5	0.1	Iris-setosa
9	4.9	3.1	1.5	0.1	Iris-setosa
12	4.8	3.0	1.4	0.1	Iris-setosa

In [92]:
```
# Sort the DataFrame by petal_width and display the NumPy array.
iris.sort_values(by='petal_width', ascending=True).values[0:5]
```

Out[92]:
```
array([[5.2, 4.1, 1.5, 0.1, 'Iris-setosa'],
       [4.3, 3.0, 1.1, 0.1, 'Iris-setosa'],
       [4.9, 3.1, 1.5, 0.1, 'Iris-setosa'],
       [4.9, 3.1, 1.5, 0.1, 'Iris-setosa'],
       [4.8, 3.0, 1.4, 0.1, 'Iris-setosa']], dtype=object)
```

Split-apply-combine: Explore the data while using a `groupby` on `'species'`.

In [93]:
```
# Mean of sepal_length, grouped by species.
iris.groupby(by='species', axis=0).sepal_length.mean()
```

Out[93]:
```
species
Iris-setosa        5.006
Iris-versicolor    5.936
Iris-virginica     6.588
Name: sepal_length, dtype: float64
```

In [94]:
```
# Mean of all numeric columns, grouped by species.
iris.groupby('species').mean()
```

Out[94]:

	sepal_length	sepal_width	petal_length	petal_width
species				
Iris-setosa	5.006	3.418	1.464	0.244
Iris-versicolor	5.936	2.770	4.260	1.326
Iris-virginica	6.588	2.974	5.552	2.026

In [95]:
```
# describe() all numeric columns, grouped by species.
iris.groupby('species').describe()
```

Out[95]:

	petal_length								petal_width		...	sepal_length		sepal_w	
	count	mean	std	min	25%	50%	75%	max	count	mean	...	75%	max	count	n
species															
Iris-setosa	50.0	1.464	0.173511	1.0	1.4	1.50	1.575	1.9	50.0	0.244	...	5.2	5.8	50.0	3
Iris-versicolor	50.0	4.260	0.469911	3.0	4.0	4.35	4.600	5.1	50.0	1.326	...	6.3	7.0	50.0	2
Iris-virginica	50.0	5.552	0.551895	4.5	5.1	5.55	5.875	6.9	50.0	2.026	...	6.9	7.9	50.0	2

3 rows × 32 columns

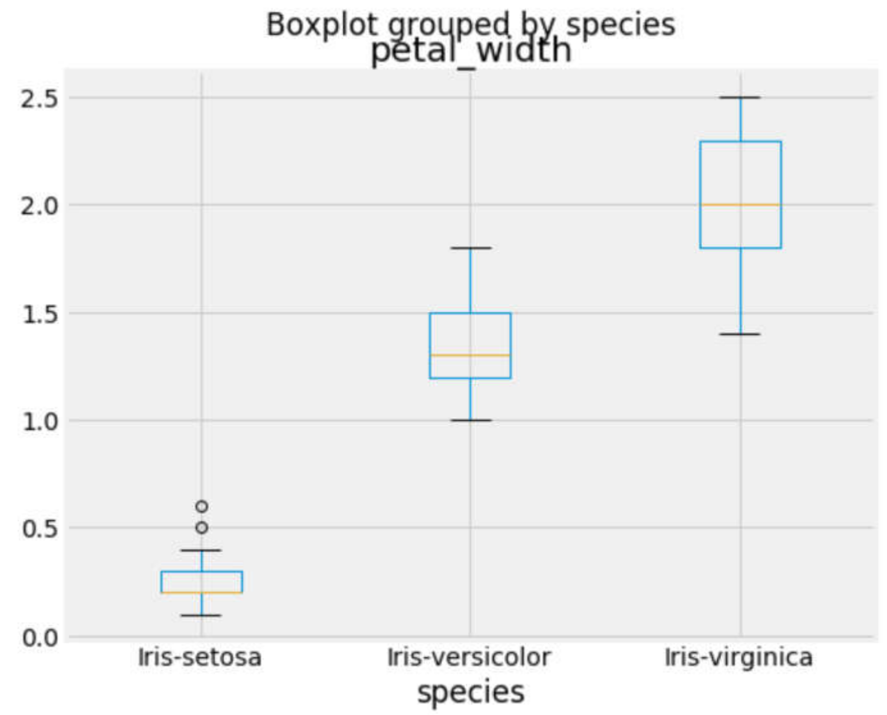
In [96]:
```
# describe() all numeric columns, grouped by species.
iris.groupby('species').describe()
```

Out[96]:

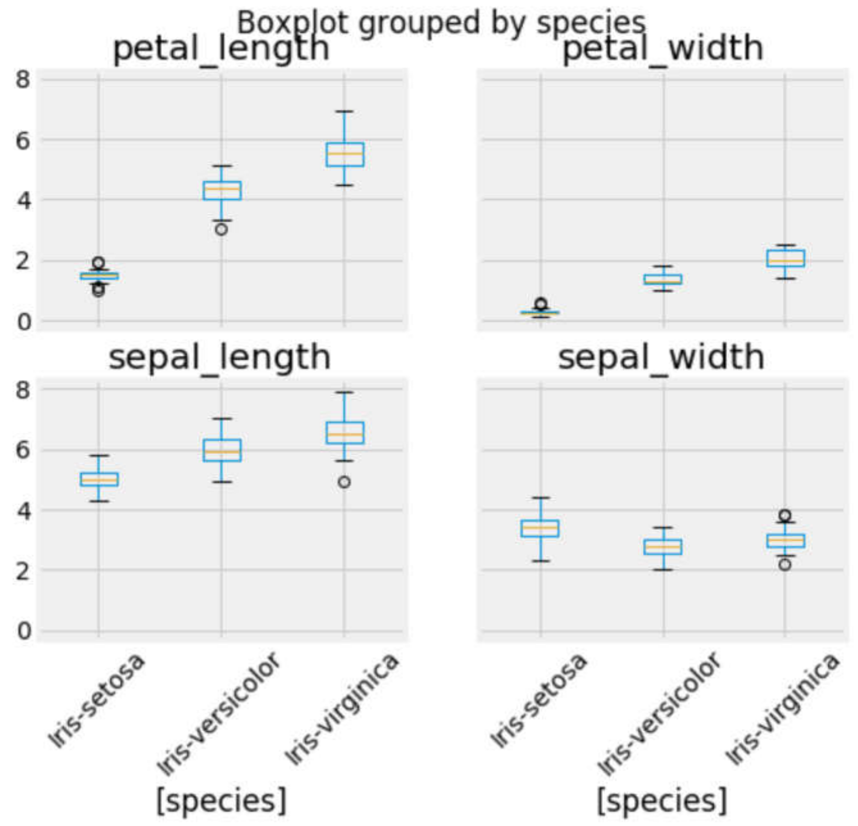
	petal_length								petal_width		...	sepal_length		sepal_w	
	count	mean	std	min	25%	50%	75%	max	count	mean	...	75%	max	count	n
species															
Iris-setosa	50.0	1.464	0.173511	1.0	1.4	1.50	1.575	1.9	50.0	0.244	...	5.2	5.8	50.0	3
Iris-versicolor	50.0	4.260	0.469911	3.0	4.0	4.35	4.600	5.1	50.0	1.326	...	6.3	7.0	50.0	2
Iris-virginica	50.0	5.552	0.551895	4.5	5.1	5.55	5.875	6.9	50.0	2.026	...	6.9	7.9	50.0	2

3 rows × 32 columns

In [97]:
```
# Box plot of petal_width, grouped by species.
iris.boxplot(column='petal_width', by='species');
```



In [98]:
```
# Box plot of all numeric columns, grouped by species.
iris.boxplot(by='species', rot=45);
```



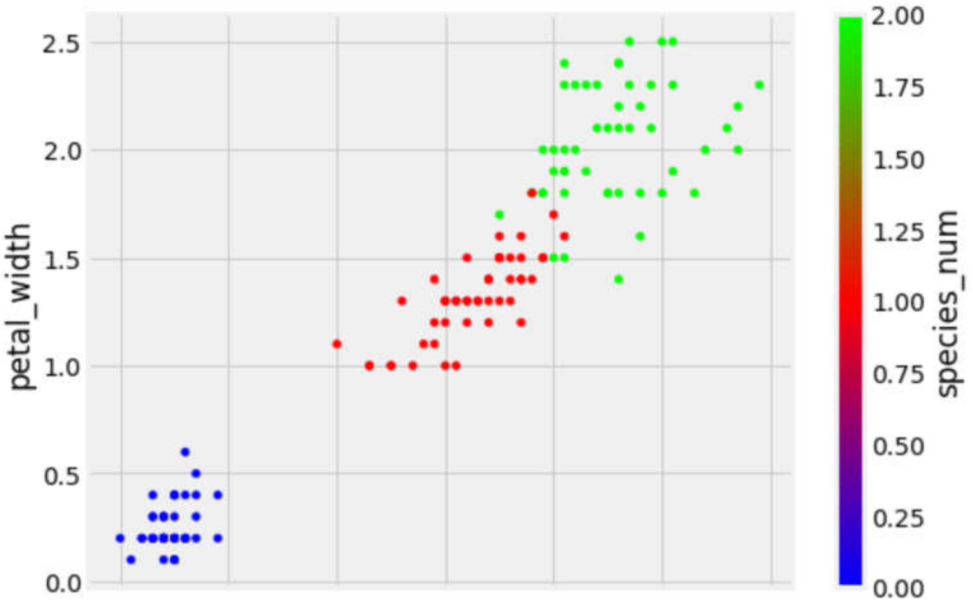
In [99]:
```
# Map species to a numeric value so that plots can be colored by species.
iris['species_num'] = iris.species.map({'Iris-setosa':0, 'Iris-versicolor':1, 'Iris
-virginica':2})

# Alternative method:
iris['species_num'] = iris.species.factorize()[0]
```

In [100]:
```
iris.head()
```

Out[100]:

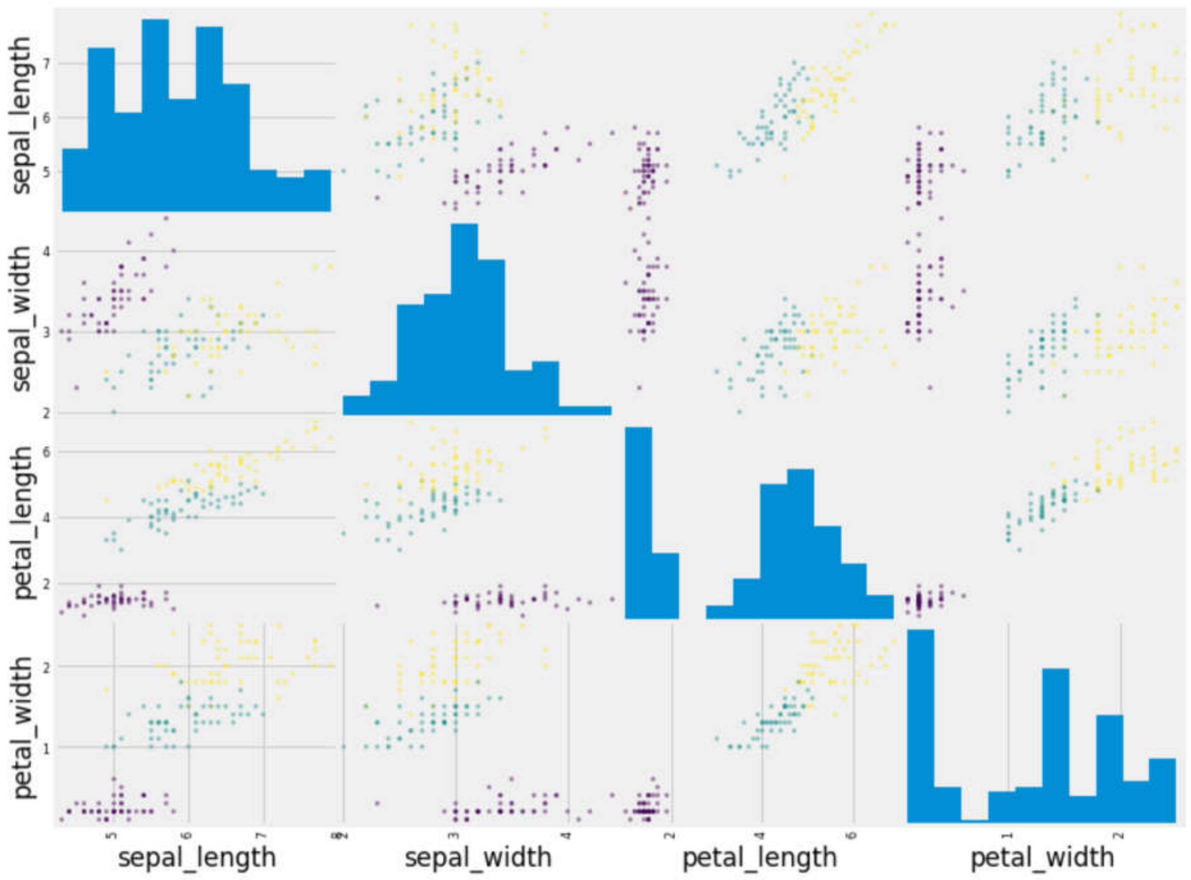
	sepal_length	sepal_width	petal_length	petal_width	species	species_num
32	5.2	4.1	1.5	0.1	Iris-setosa	0
13	4.3	3.0	1.1	0.1	Iris-setosa	0
37	4.9	3.1	1.5	0.1	Iris-setosa	0
9	4.9	3.1	1.5	0.1	Iris-setosa	0
12	4.8	3.0	1.4	0.1	Iris-setosa	0

In [101]:
```
# Scatterplot of petal_length vs. petal_width, colored by species
iris.plot(kind='scatter', x='petal_length', y='petal_width', c='species_num', colo
rmap='brg');
```



In [102]:
```
# Scatter matrix of all features, colored by species.
pd.tools.plotting.scatter_matrix(iris.drop('species_num', axis=1), c=iris.species_
num, figsize=(12, 10));
```

```
C:\ProgramData\Anaconda3\lib\site-packages\ipykernel_launcher.py:2: FutureWarnin
g: 'pandas.tools.plotting.scatter_matrix' is deprecated, import 'pandas.plotting
.scatter_matrix' instead.
```



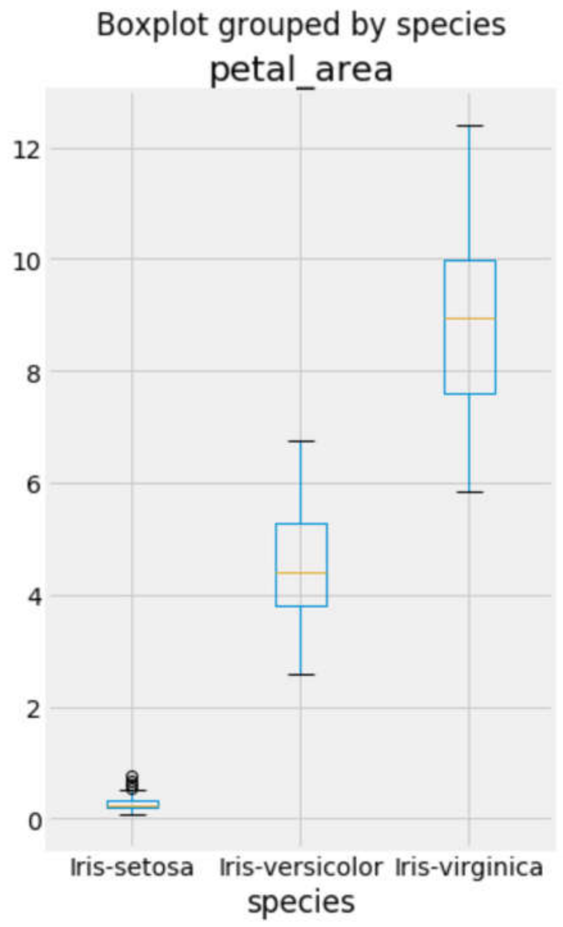
Write down a set of rules that could be used to predict species based on iris measurements.

In [103]:
```
# Define a new feature that represents petal area ("feature engineering").
# As iris petals are more ovular shaped as opposed to rectangular,
# we're going to use the formula for area of an ellipse:
# r1 * r2 * 3.14.
iris['petal_area'] = ((iris.petal_length/2) * (iris.petal_width/2) * 3.124)
```

In [104]:
```
# Description of petal_area, grouped by species.
iris.groupby('species').petal_area.describe().unstack()
```

Out[104]:
```
       species
count  Iris-setosa        50.000000
       Iris-versicolor    50.000000
       Iris-virginica     50.000000
mean   Iris-setosa         0.283347
       Iris-versicolor     4.467632
       Iris-virginica      8.822332
std    Iris-setosa         0.143117
       Iris-versicolor     1.068723
       Iris-virginica      1.684939
min    Iris-setosa         0.085910
       Iris-versicolor     2.577300
       Iris-virginica      5.857500
25%    Iris-setosa         0.206965
       Iris-versicolor     3.795660
       Iris-virginica      7.589368
50%    Iris-setosa         0.234300
       Iris-versicolor     4.385315
       Iris-virginica      8.938545
75%    Iris-setosa         0.328020
       Iris-versicolor     5.271750
       Iris-virginica      9.988990
max    Iris-setosa         0.749760
       Iris-versicolor     6.747840
       Iris-virginica     12.394470
dtype: float64
```

In [105]:
```
# Box plot of petal_area, grouped by species.
iris.boxplot(column='petal_area', by='species',figsize=(5,8));
```



```
In [106]: # Only show irises with a petal_area between 3 and 7.
          iris[(iris.petal_area > 3) & (iris.petal_area < 7)].sort_values('petal_area')
```

Out[106]:

	sepal_length	sepal_width	petal_length	petal_width	species	species_num	petal_area
62	6.0	2.2	4.0	1.0	Iris-versicolor	1	3.12400
67	5.8	2.7	4.1	1.0	Iris-versicolor	1	3.20210
80	5.5	2.4	3.8	1.1	Iris-versicolor	1	3.26458
69	5.6	2.5	3.9	1.1	Iris-versicolor	1	3.35049
82	5.8	2.7	3.9	1.2	Iris-versicolor	1	3.65508
64	5.6	2.9	3.6	1.3	Iris-versicolor	1	3.65508
92	5.8	2.6	4.0	1.2	Iris-versicolor	1	3.74880
95	5.7	3.0	4.2	1.2	Iris-versicolor	1	3.93624
53	5.5	2.3	4.0	1.3	Iris-versicolor	1	4.06120
89	5.5	2.5	4.0	1.3	Iris-versicolor	1	4.06120
71	6.1	2.8	4.0	1.3	Iris-versicolor	1	4.06120
90	5.5	2.6	4.4	1.2	Iris-versicolor	1	4.12368
88	5.6	3.0	4.1	1.3	Iris-versicolor	1	4.16273
99	5.7	2.8	4.1	1.3	Iris-versicolor	1	4.16273
59	5.2	2.7	3.9	1.4	Iris-versicolor	1	4.26426
94	5.6	2.7	4.2	1.3	Iris-versicolor	1	4.26426
96	5.7	2.9	4.2	1.3	Iris-versicolor	1	4.26426
97	6.2	2.9	4.3	1.3	Iris-versicolor	1	4.36579
74	6.4	2.9	4.3	1.3	Iris-versicolor	1	4.36579
73	6.1	2.8	4.7	1.2	Iris-versicolor	1	4.40484
87	6.3	2.3	4.4	1.3	Iris-versicolor	1	4.46732
55	5.7	2.8	4.5	1.3	Iris-versicolor	1	4.56885
58	6.6	2.9	4.6	1.3	Iris-versicolor	1	4.67038
65	6.7	3.1	4.4	1.4	Iris-versicolor	1	4.81096
75	6.6	3.0	4.4	1.4	Iris-versicolor	1	4.81096
61	5.9	3.0	4.2	1.5	Iris-versicolor	1	4.92030
91	6.1	3.0	4.6	1.4	Iris-versicolor	1	5.02964
63	6.1	2.9	4.7	1.4	Iris-versicolor	1	5.13898
50	7.0	3.2	4.7	1.4	Iris-versicolor	1	5.13898
76	6.8	2.8	4.8	1.4	Iris-versicolor	1	5.24832
68	6.2	2.2	4.5	1.5	Iris-versicolor	1	5.27175
66	5.6	3.0	4.5	1.5	Iris-versicolor	1	5.27175
78	6.0	2.9	4.5	1.5	Iris-versicolor	1	5.27175
84	5.4	3.0	4.5	1.5	Iris-versicolor	1	5.27175
51	6.4	3.2	4.5	1.5	Iris-versicolor	1	5.27175
54	6.5	2.8	4.6	1.5	Iris-versicolor	1	5.38890
86	6.7	3.1	4.7	1.5	Iris-versicolor	1	5.50605

My set of rules for predicting species:

- If petal_area is less than 2, predict **setosa**.
- Else if petal_area is less than 7.4, predict **versicolor**.
- Otherwise, predict **virginica**.

Bonus: Define a function that accepts a row of data and returns a predicted species. Then, use that function to make predictions for all existing rows of data and check the accuracy of your predictions.

In [107]:
```
val_a,val_b = ('a','b')
val_a
```

Out[107]:
```
'a'
```

In [108]:
```
def predict_flower(df):
    preds = []
    for ind, row in df.iterrows():
        if row.petal_area < 2:
            prediction = 'Iris-setosa'
        elif row.petal_area < 7.4:
            prediction = 'Iris-versicolor'
        else:
            prediction = 'Iris-virginica'
        preds.append(prediction)

    df['prediction'] = preds


predict_flower(iris)
```

In [109]:
```
iris.head()
```

Out[109]:

	sepal_length	sepal_width	petal_length	petal_width	species	species_num	petal_area	predictio
32	5.2	4.1	1.5	0.1	Iris-setosa	0	0.11715	Iris-setosa
13	4.3	3.0	1.1	0.1	Iris-setosa	0	0.08591	Iris-setosa
37	4.9	3.1	1.5	0.1	Iris-setosa	0	0.11715	Iris-setosa
9	4.9	3.1	1.5	0.1	Iris-setosa	0	0.11715	Iris-setosa
12	4.8	3.0	1.4	0.1	Iris-setosa	0	0.10934	Iris-setosa

In [110]:
```
# 0.3333 means 1/3 are classified correctly

sum(iris.species == iris.prediction) / 150.
```

Out[110]:
```
0.9333333333333335
```

Human Learning on the Iris Data Set

How did we (as humans) predict the species of an iris?

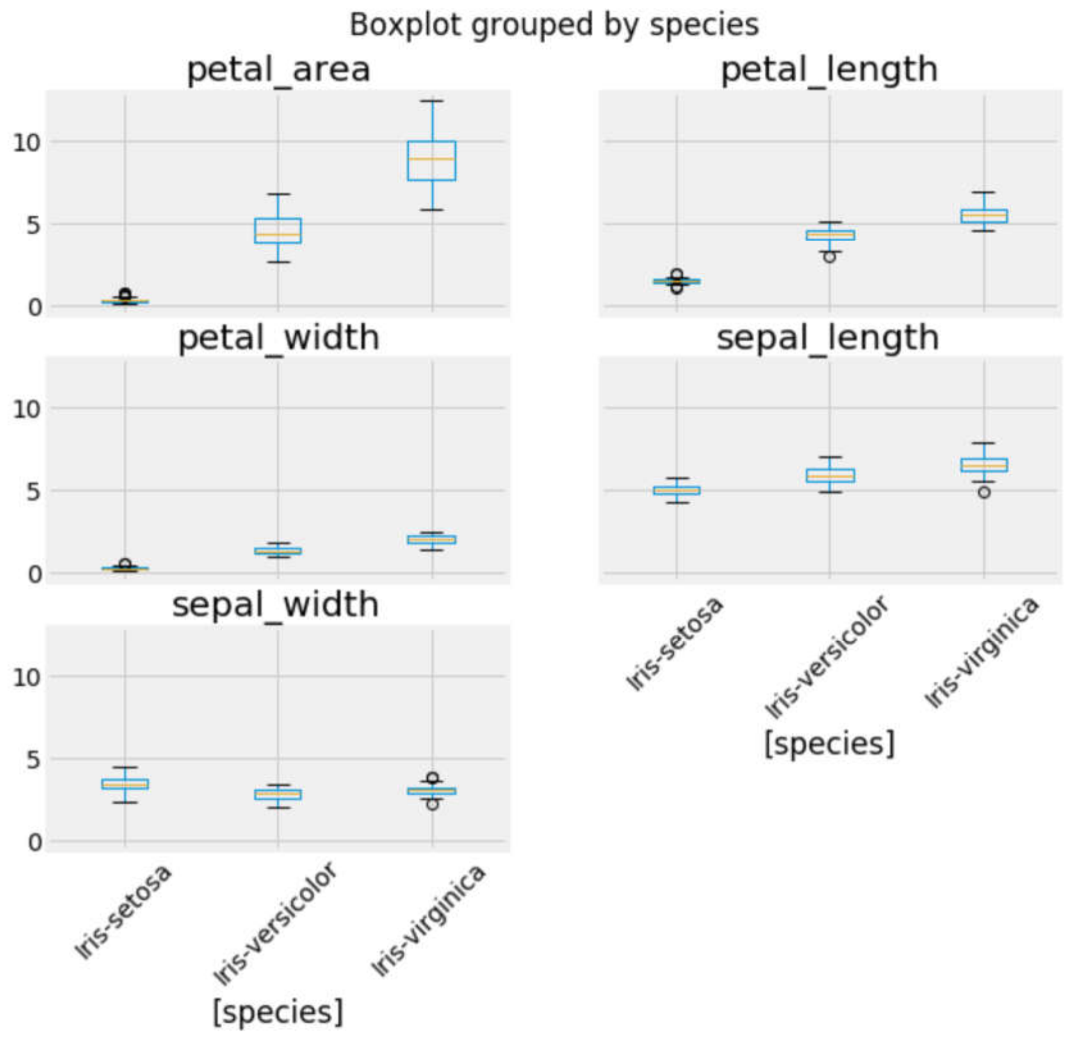
1. We observed that the different species had (somewhat) dissimilar measurements.
2. We focused on features that seemed to correlate with the response.
3. We created a set of rules (using those features) to predict the species of an unknown iris.

We assumed that if an **unknown iris** had measurements similar to **previous irises**, then its species was most likely the same as those previous irises.

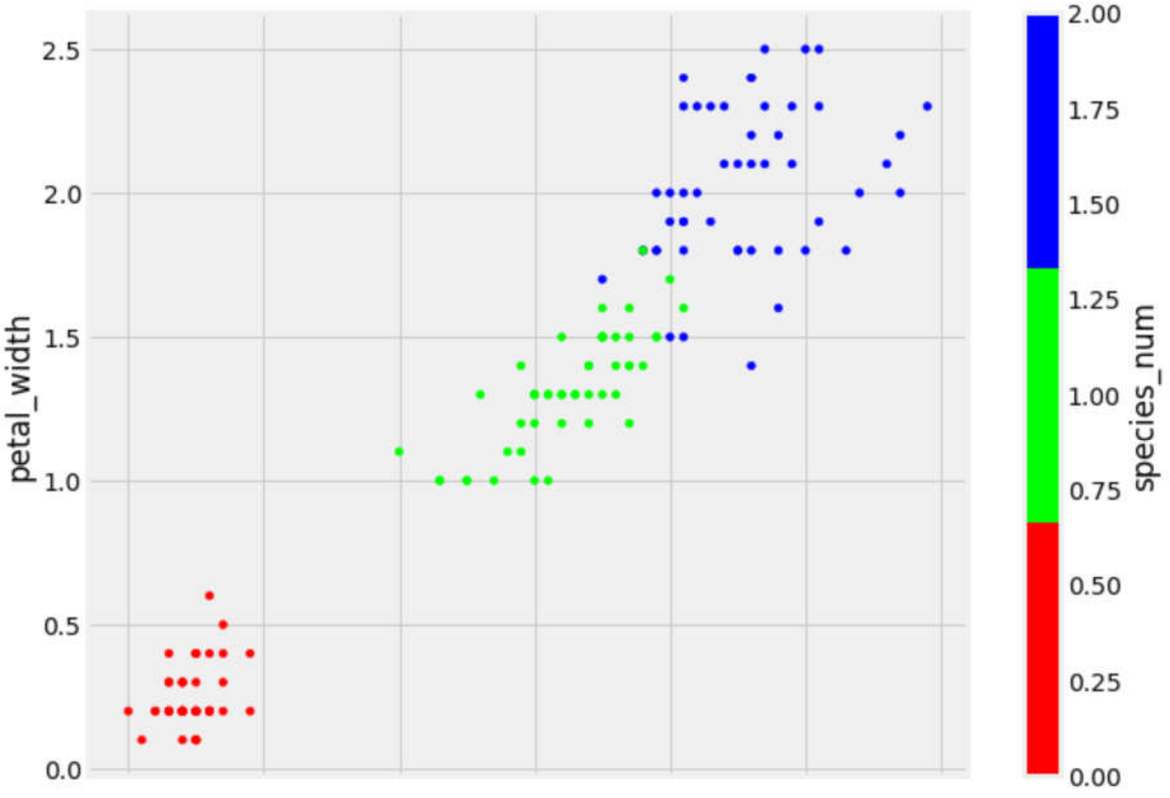
In [111]:
```
# Allow plots to appear in the notebook.
%matplotlib inline
import matplotlib.pyplot as plt

# Increase default figure and font sizes for easier viewing.
plt.rcParams['figure.figsize'] = (10, 8)
plt.rcParams['font.size'] = 14

# Create a custom color map.
from matplotlib.colors import ListedColormap
cmap_bold = ListedColormap(['#FF0000', '#00FF00', '#0000FF'])
```

In [112]:
```
# Map each iris species to a number.
iris['species_num'] = iris.species.map({'Iris-setosa':0, 'Iris-versicolor':1, 'Iri
s-virginica':2})
```

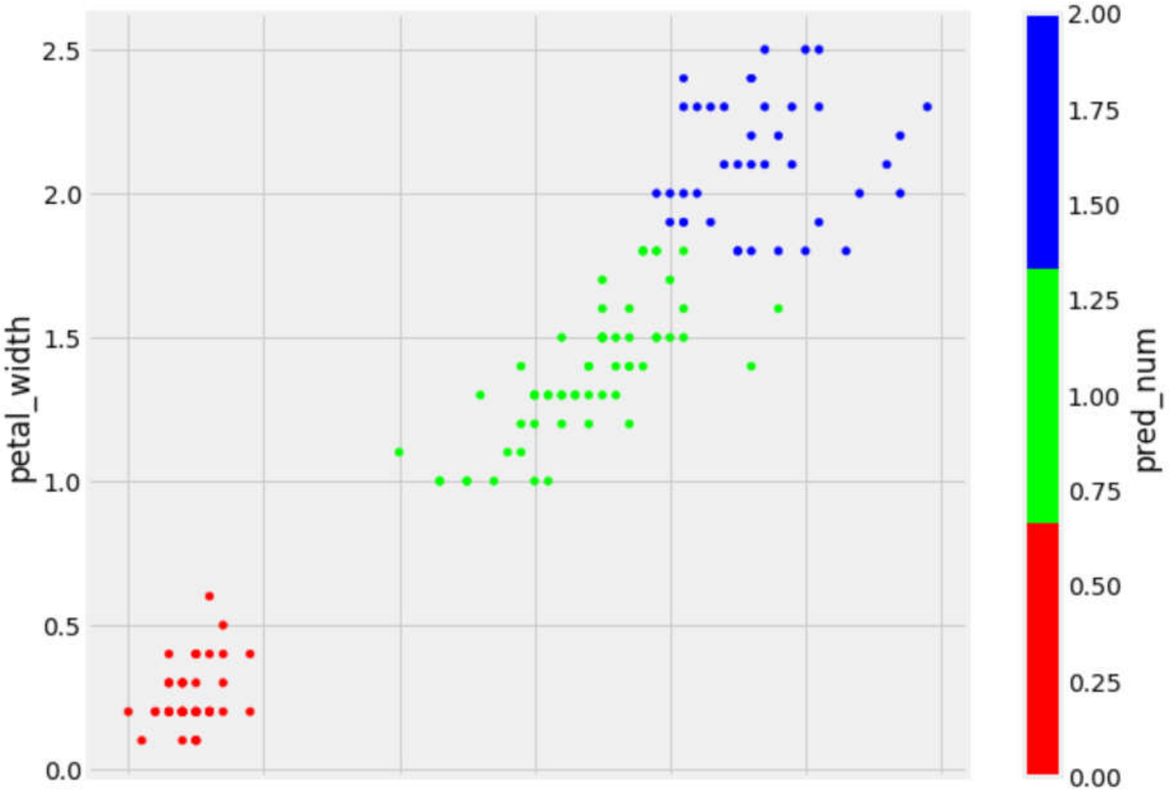
In [113]:
```
# Box plot of all numeric columns, grouped by species.
iris.drop('species_num', axis=1).boxplot(by='species', rot=45);
```



```
In [114]: # Create a scatterplot of PETAL LENGTH versus PETAL WIDTH and color by SPECIES.
          iris.plot(kind='scatter', x='petal_length', y='petal_width', c='species_num', colo
          rmap=cmap_bold);
```



In [115]:
```
iris['pred_num'] = iris.prediction.map({'Iris-setosa':0, 'Iris-versicolor':1, 'Iri
s-virginica':2})



# Create a scatterplot of PETAL LENGTH versus PETAL WIDTH and color by PREDICTION.
iris.plot(kind='scatter', x='petal_length', y='petal_width', c='pred_num', colorma
p=cmap_bold);
```



K-Nearest Neighbors (KNN) Classification

K-nearest neighbors classification is (as its name implies) a classification model that uses the "K" most similar observations in order to make a prediction.

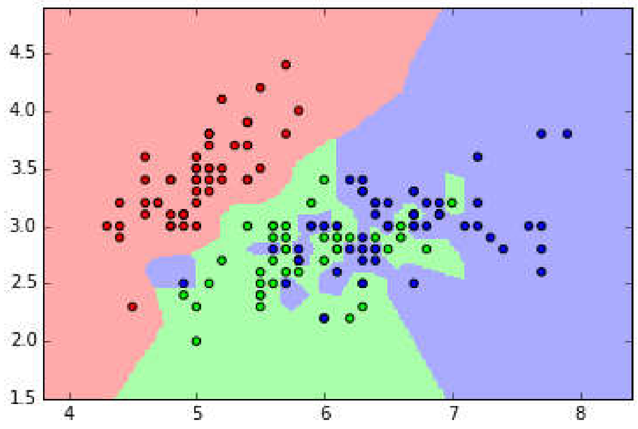
KNN is a supervised learning method; therefore, the training data must have known target values.

The process of of prediction using KNN is fairly straightforward:

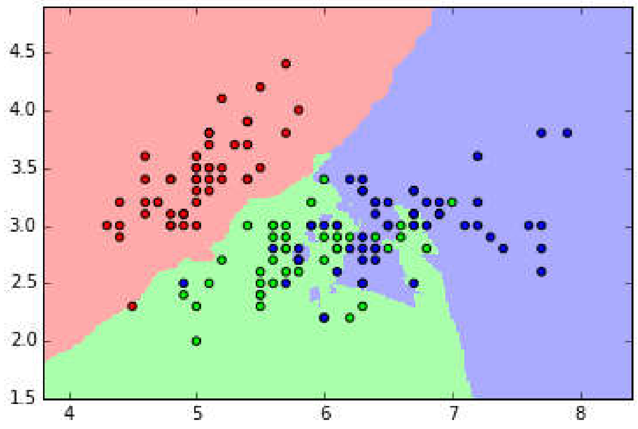
1. Pick a value for K.
2. Search for the K observations in the data that are "nearest" to the measurements of the unknown iris.
    - Euclidian distance is often used as the distance metric, but other metrics are allowed.
3. Use the most popular response value from the K "nearest neighbors" as the predicted response value for the unknown iris.

The visualizations below show how a given area can change in its prediction as K changes. Colored points represent true values and colored areas represent a **prediction space**. If an unknown point was to fall in a space, its predicted value would be the color of that scace.

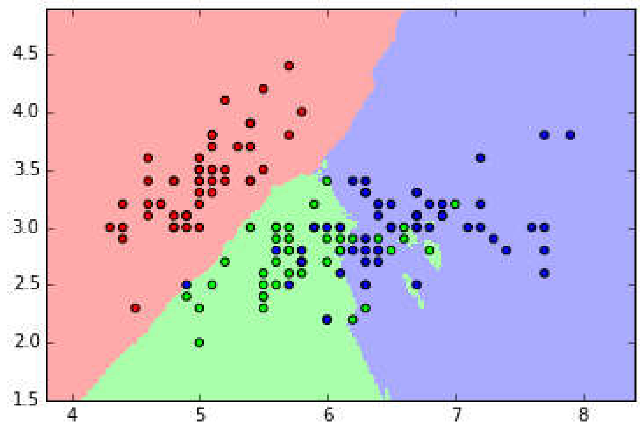
KNN Classification Map for Iris (K=1)



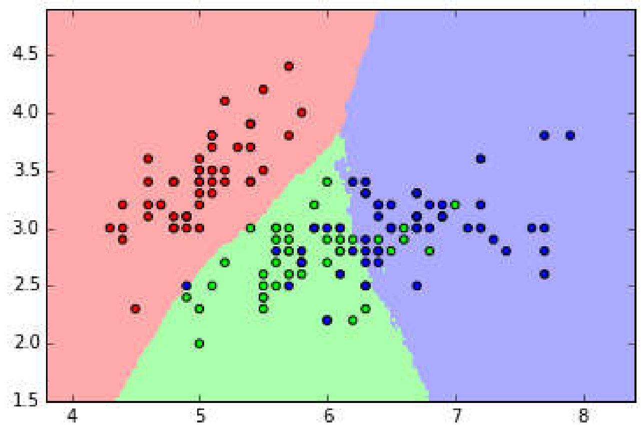
KNN Classification Map for Iris (K=5)



KNN Classification Map for Iris (K=15)



KNN Classification Map for Iris (K=50)



We can see that, as more Ks are added, the classification spaces' borders become more distinct. However, you can also see that the spaces are not perfectly pure when it comes to the known elements within them.

How are outliers affected by K? As K increases, outliers are "smoothed out". Look at the above three plots and notice how outliers strongly affect the prediction space when K=1. When K=50, outliers no longer affect region boundaries. This is a classic bias-variance tradeoff -- with increasing K, the bias increases but the variance decreases.

Question: What's the "best" value for K in this case?

Answer: The value which produces the most accurate predictions on **unseen data**. We want to create a model that generalizes!

For the rest of the lesson, we will be using a dataset containing the 2015 season statistics for ~500 NBA players. This dataset leads to a nice choice of K, as we'll see below. Its columns are:

Column	Meaning
pos	C: Center. F: Front. G: Guard
ast	
blk	
tov	
pf	

In [116]:
```
# Read the NBA data into a DataFrame.
import pandas as pd

path = 'NBA_players_2015.csv'
nba = pd.read_csv(path, index_col=0)
```

In [117]:
```
# Map positions to numbers.
nba['pos_num'] = nba.pos.map({'C':0, 'F':1, 'G':2})
```

In [118]:
```
# Create feature matrix (X).
feature_cols = ['ast', 'stl', 'blk', 'tov', 'pf']
X = nba[feature_cols]
```

In [119]:
```
# Create response vector (y).
y = nba.pos_num
```

Using the Train/Test Split Procedure (K=1)

In [120]:
```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import train_test_split
from sklearn import metrics
```

Step 1: Split X and y into training and testing sets (using `random_state` for reproducibility).

In [121]:
```
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=99)
```

Step 2: Train the model on the training set (using K=1).

In [122]:
```
knn = KNeighborsClassifier(n_neighbors=1)
knn.fit(X_train, y_train)
```

Out[122]:
```
KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
           metric_params=None, n_jobs=1, n_neighbors=1, p=2,
           weights='uniform')
```

Step 3: Test the model on the testing set and check the accuracy.

```
In [123]: y_pred_class = knn.predict(X_test)
          print(metrics.accuracy_score(y_test, y_pred_class))

          0.616666666667
```

Repeating for K=50.

```
In [124]: knn = KNeighborsClassifier(n_neighbors=50)
          knn.fit(X_train, y_train)
          y_pred_class = knn.predict(X_test)
          print(metrics.accuracy_score(y_test, y_pred_class))

          0.675
```

Comparing Testing Accuracy With Null Accuracy

Null accuracy is the accuracy that can be achieved by **always predicting the most frequent class**. It is a benchmark against which you may want to measure every classification model.

Examine the class distribution.

```
In [125]: y_test.value_counts()

Out[125]: 2    60
          1    49
          0    11
          Name: pos_num, dtype: int64
```

Compute null accuracy.

```
In [126]: y_test.value_counts().head(1) / len(y_test)

Out[126]: 2    0.5
          Name: pos_num, dtype: float64
```

Tuning a KNN Model

```
In [127]: # Instantiate the model (using the value K=5).
          knn = KNeighborsClassifier(n_neighbors=5)

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

          # Store the predicted response values.
          y_pred_class = knn.predict(X)
```

Question: Which model produced the correct predictions for the two unknown positions?

Answer: We don't know, because these are **out-of-sample observations**, meaning that we don't know the true response values. Our goal with supervised learning is to build models that generalize to out-of-sample data. However, we can't truly measure how well our models will perform on out-of-sample data.

Question: Does that mean that we have to guess how well our models are likely to do?

Answer: Thankfully, no. We'll discuss **model evaluation procedures**, which allow us to use our existing labeled data to estimate how well our models are likely to perform on out-of-sample data. These procedures will help us to tune our models and choose between different types of models.

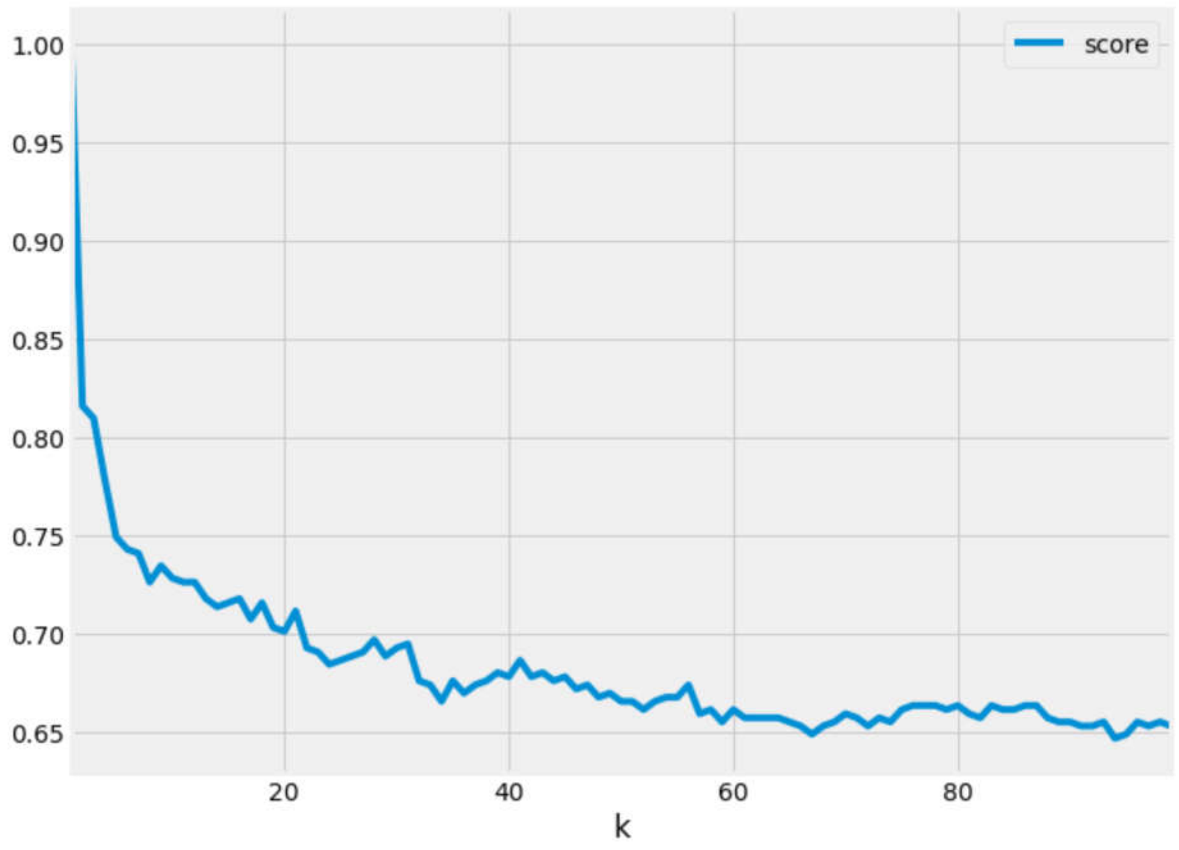
In [128]:
```
# Calculate predicted probabilities of class membership.
knn.predict_proba(X)
```

Out[128]:
```
array([[ 0. ,  0.6,  0.4],
       [ 0. ,  0.8,  0.2],
       [ 0.8,  0.2,  0. ],
       ...,
       [ 0. ,  0.6,  0.4],
       [ 0.6,  0.4,  0. ],
       [ 0.6,  0.4,  0. ]])
```

What Happens If We View the Accuracy of our Training Data?

In [129]:
```
scores = []
for k in range(1,100):
    knn = KNeighborsClassifier(n_neighbors=k)
    knn.fit(X,y)
    pred = knn.predict(X)
    score = float(sum(pred == y)) / len(y)
    scores.append([k, score])
```

In [130]:
```
data = pd.DataFrame(scores,columns=['k','score'])
data.plot.line(x='k',y='score');
```



Search for the "best" value of K.

In [131]:
```
# Calculate TRAINING ERROR and TESTING ERROR for K=1 through 100.

k_range = range(1, 101)
training_error = []
testing_error = []

# Find test accuracy for all values of K between 1 and 100 (inclusive).
for k in k_range:

    # Instantiate the model with the current K value.
    knn = KNeighborsClassifier(n_neighbors=k)
    knn.fit(X_train, y_train)

    # Calculate training error (error = 1 - accuracy).
    y_pred_class = knn.predict(X)
    training_accuracy = metrics.accuracy_score(y, y_pred_class)
    training_error.append(1 - training_accuracy)

    # Calculate testing error.
    y_pred_class = knn.predict(X_test)
    testing_accuracy = metrics.accuracy_score(y_test, y_pred_class)
    testing_error.append(1 - testing_accuracy)
```

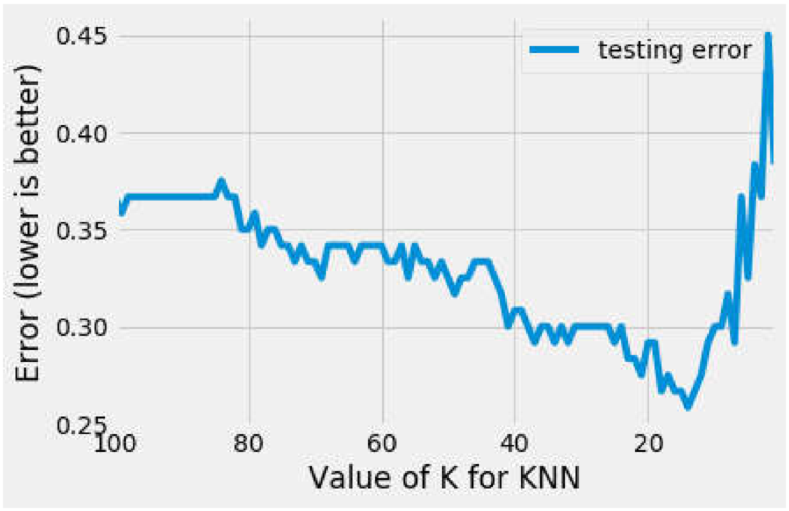
In [132]:
```
# Allow plots to appear in the notebook.
%matplotlib inline
import matplotlib.pyplot as plt
plt.style.use('fivethirtyeight')
```

In [133]:
```
# Create a DataFrame of K, training error, and testing error.
column_dict = {'K': k_range, 'training error':training_error, 'testing error':test
ing_error}
df = pd.DataFrame(column_dict).set_index('K').sort_index(ascending=False)
df.head()
```

Out[133]:

	testing error	training error
K		
100	0.366667	0.382845
99	0.358333	0.378661
98	0.366667	0.384937
97	0.366667	0.384937
96	0.366667	0.380753

In [134]:
```
# Plot the relationship between K (HIGH TO LOW) and TESTING ERROR.
df.plot(y='testing error');
plt.xlabel('Value of K for KNN');
plt.ylabel('Error (lower is better)');
```



In [135]:
```
# Find the minimum testing error and the associated K value.
df.sort_values('testing error').head()
```

Out[135]:

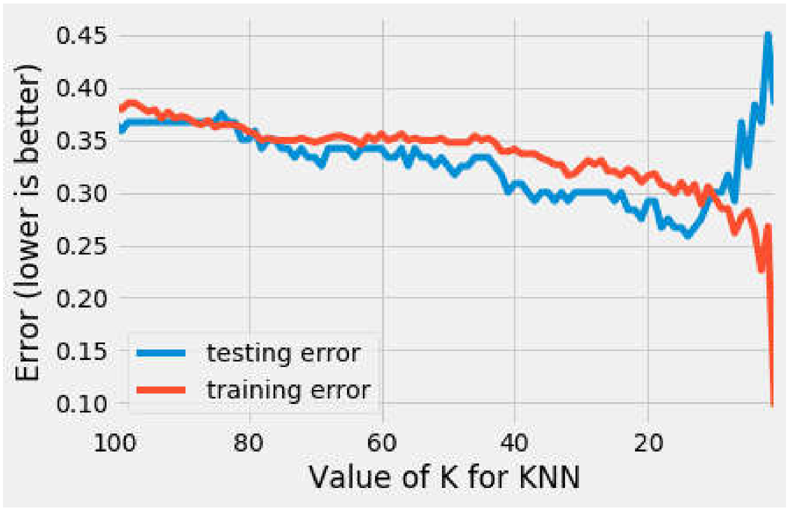
	testing error	training error
K		
14	0.258333	0.299163
13	0.266667	0.307531
18	0.266667	0.307531
16	0.266667	0.299163
15	0.266667	0.309623

In [136]:
```
# Alternative method:
min(zip(testing_error, k_range))
```

Out[136]:
```
(0.2583333333333333, 14)
```

Training Error Versus Testing Error

In [137]:
```
# Plot the relationship between K (HIGH TO LOW) and both TRAINING ERROR and TESTIN
G ERROR.
df.plot();
plt.xlabel('Value of K for KNN');
plt.ylabel('Error (lower is better)');
```



- **Training error** decreases as model complexity increases (lower value of K).
- **Testing error** is minimized at the optimum model complexity.

Evaluating the training and testing error is important. For example:

- If the training error is much lower than the test error, then our model is likely overfitting.
- If the test error starts increasing as we vary a hyperparameter, we may be overfitting.
- If either error plateaus, our model is likely underfitting (not complex enough).

Making Predictions on Out-of-Sample Data

Given the statistics of a (truly) unknown NBA player, how do we predict his position?

```
In [138]: import numpy as np

          # Instantiate the model with the best-known parameters.
          knn = KNeighborsClassifier(n_neighbors=14)

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

          # Make a prediction for an out-of-sample observation.
          knn.predict(np.array([2, 1, 0, 1, 2]).reshape(1, -1))
Out[138]: array([2], dtype=int64)
```

What could we conclude?

- When using KNN on this data set with these features, the **best value for K** is likely to be around 14.
- Given the statistics of an **unknown player**, we estimate that we would be able to correctly predict his position about 74% of the time.

Let's do one more example with cross-validation

Rather than reporting testing error, we will now report testing accuracy which is $1 - testingerror$.

```
In [139]: from sklearn.model_selection import cross_val_score

          # This will overwrite our previous instance of KN
          knn = KNeighborsClassifier(n_neighbors=14)

          # It is possible to do cross-validation in a manual manner,
          # but this is a frequent shortcut
          cross_val_score(knn, X, y, cv = 5, scoring='accuracy')
Out[139]: array([ 0.67708333,  0.625     ,  0.6875    ,  0.66666667,  0.70212766])
```

By default, cv is the number of folds using the KFold object.

However cv will also accept a cross validation iterator such as KFolds, ShuffleSplit, or TimeSeriesSplit.

Described here:

http://scikit-learn.org/stable/modules/cross_validation.html#computing-cross-validated-metrics (http://scikit-learn.org/stable/modules/cross_validation.html#computing-cross-validated-metrics)

A full list of cross validation iterators:

http://scikit-learn.org/stable/modules/classes.html#splitter-classes (http://scikit-learn.org/stable/modules/classes.html#splitter-classes)

Average results from the folds to get an interpretable score

```
In [140]: np.mean(cross_val_score(knn,X, y, cv = 5, scoring='accuracy'))
Out[140]: 0.6716755319148936
```

Standardizing Features

There is one major issue that applies to many machine learning models: They are sensitive to feature scale.

> KNN in particular is sensitive to feature scale because it (by default) uses the Euclidean distance metric. To determine closeness, Euclidean distance sums the square difference along each axis. So, if one axis has large differences and another has small differences, the former axis will contribute much more to the distance than the latter axis.

This means that it matters whether our feature are centered around zero and have similar variance to each other.

In the case of KNN on the iris data set, imagine we measure sepal length in kilometers, but we measure sepal width in millimeters. Our data will show variation in sepal width, but almost no variation in sepal length.

Unfortunately, KNN cannot automatically adjust to this. Other models tend to struggle with scale as well, even linear regression, when you get into more advanced methods such as regularization.

Fortuantely, this is an easy fix.

Use `StandardScaler` to Standardize our Data

StandardScaler standardizes our data by subtracting the mean from each feature and dividing by its standard deviation.

Separate feature matrix and response for scikit-learn.

```
In [141]: # Create feature matrix (X).
          feature_cols = ['ast', 'stl', 'blk', 'tov', 'pf']

          X = nba[feature_cols]
          y = nba.pos_num  # Create response vector (y).
```

Create the train/test split.

Notice that we create the train/test split first. This is because we will reveal information about our testing data if we standardize right away.

```
In [142]: X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=99)
```

Instantiate and fit `StandardScaler`.

```
In [143]: from sklearn.preprocessing import StandardScaler

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

Fit a KNN model and look at the testing error.

Can you find a number of neighbors that improves our results from before?

```
In [144]: # Calculate testing error.
          knn = KNeighborsClassifier(n_neighbors=12)
          knn.fit(X_train, y_train)

          y_pred_class = knn.predict(X_test)
          testing_accuracy = metrics.accuracy_score(y_test, y_pred_class)
          testing_error = 1 - testing_accuracy

          print (testing_accuracy)

          0.725
```

Practice with Iris data

So far we've seen a lot a details about the machine learning workflow. Let's make sure we cna put the pieces together making a model to predict irises types based off of the iris data.

Like a real project, there is no prompt. Unlike a real project, this data is very easy to work with and does not require further exploration.

The deliverable is a knn instance that can predict irises using sepal width and length and petal length and width.

```
In [145]: # Read the iris data into a DataFrame.
          import pandas as pd
          import numpy as np

          data = 'iris.data'
          iris = pd.read_csv(data)
```

```
In [146]: iris.columns

Out[146]: Index(['sepal_length', 'sepal_width', 'petal_length', 'petal_width',
                 'species'],
                dtype='object')
```

```
In [147]: features = ['sepal_length', 'sepal_width', 'petal_length', 'petal_width']
          X = iris[features]
          y = iris['species']
```

```
In [148]: knn = KNeighborsClassifier(n_neighbors=10)
          np.mean(cross_val_score(knn, X, y, cv=5, scoring='accuracy'))

Out[148]: 0.98000000000000009
```

Anywhere between 10 and 12 neighbors is probably a good selection.

Did you try scaling the data? Did you make sure to use fit_transform for training data and transform for the testing data?

If you want to see how to scale using cross_validation, there's an example below:

```
In [149]: from sklearn.pipeline import Pipeline
          # Pipeline allows use to combine preprocessing operations and modeling into one st
          ep
          clf = Pipeline([('transformer', StandardScaler), ('knn', knn)])
          np.mean(cross_val_score(knn, X, y, cv=5, scoring='accuracy'))

Out[149]: 0.98000000000000009
```

If iris classification is too easy, we can try some Optical Character Recognition

Optical Character Recognition is the machine task of recognizing characters, in this case, hand-written digits.

We'll will use a famous dataset, MNIST digits, to see what we can do with KNN. The resolution is very low and only shows 8 by 8 pixel images.

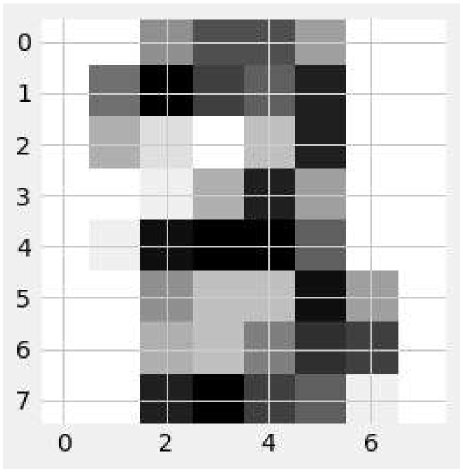
The dataset and description is available here: http://archive.ics.uci.edu/ml/datasets/optical+recognition+of+handwritten+digits (http://archive.ics.uci.edu/ml/datasets/optical+recognition+of+handwritten+digits)

Some of the hard work has already been done for us. The data is split into train and test. It has also been preprocessed to remove common irregularities.

```
In [150]: digits = pd.read_csv('http://archive.ics.uci.edu/ml/machine-learning-databases/opt
          digits/optdigits.tra', header=None)
          X = digits.iloc[:,0:64]
          y = digits.iloc[:,64]
```

```
In [151]: def plot_row_image(row):
              """A helper function to reshape 64 element rows into 8 by 8 images"""
              plt.imshow(row.values.reshape([8,8]), cmap=plt.cm.gray_r)
```

```
In [152]: plot_row_image(X.iloc[14,:])
```



Build and test a KNN classifer here

```
In [153]: knn = KNeighborsClassifier(n_neighbors=5)
          np.mean(cross_val_score(knn, X, y, cv=5, scoring='accuracy'))

Out[153]: 0.9822123386818582

In [154]: knn.fit(X, y)

Out[154]: KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
                     metric_params=None, n_jobs=1, n_neighbors=5, p=2,
                     weights='uniform')
```

Comparing KNN With Other Models

Advantages of KNN:

- It's simple to understand and explain.
- Model training is fast.
- It can be used for classification and regression (for regression, take the average value of the K nearest points!).
- Being a non-parametric method, it is often successful in classification situations where the decision boundary is very irregular.

Disadvantages of KNN:

- It must store all of the training data.
- Its prediction phase can be slow when n is large.
- It is sensitive to irrelevant features.
- It is sensitive to the scale of the data.
- Accuracy is (generally) not competitive with the best supervised learning methods.

Sudmission

Please submit this file to dropbox via link:

```
- www.dropbox.com/request/IQ4kZoZDTgHr8FGoHwMA
```

Deadline: Before 10 am next Tuesday (15/05/2018)