In [ ]:
```
from datascience import *
import numpy as np
import matplotlib
from mpl_toolkits.mplot3d import Axes3D
%matplotlib inline
import matplotlib.pyplot as plots
plots.style.use('fivethirtyeight')
```

Classification

In [ ]:
```
ckd = Table.read_table('ckd.csv').relabeled('Blood Glucose Random', 'Glucose')
ckd.show(3)
```

In [ ]:
```
ckd.select('Glucose','White Blood Cell Count', 'Hemoglobin','Class').show(3)
```

In [ ]:
```
ckd.group('Class')
```

In [ ]:
```
ckd.scatter('White Blood Cell Count', 'Glucose', group = 'Class')
```

In [ ]:
```
ckd.scatter('Hemoglobin', 'Glucose', group = 'Class')
```

In [ ]:
```
banknotes = Table.read_table('banknote.csv')
banknotes
```

In [ ]:
```
banknotes.scatter('WaveletVar', 'WaveletCurt', group = 'Class')
```

In [ ]:
```
banknotes.scatter('WaveletSkew', 'Entropy', group = 'Class')
```

In [ ]:
```
fig = plots.figure(figsize=(8,8))
ax = Axes3D(fig)
ax.scatter(banknotes.column('WaveletSkew'),
```

```
              banknotes.column('WaveletVar'),
              banknotes.column('WaveletCurt'),
              c=banknotes.column('Class'),
              cmap='viridis',
             s=50);
```

In [ ]:

```
patients = Table.read_table('breast-cancer.csv').drop('ID')
patients.show(5)
```

In [ ]:

```
patients.scatter('Bland Chromatin', 'Single Epithelial Cell Size', group = 'Class')
```

In [ ]:

```
def randomize_column(a):
    return a + np.random.normal(0.0, 0.09, size=len(a))

jittered = Table().with_columns([
        'Bland Chromatin (jittered)',
        randomize_column(patients.column('Bland Chromatin')),
        'Single Epithelial Cell Size (jittered)',
        randomize_column(patients.column('Single Epithelial Cell Size')),
        'Class',
        patients.column('Class')
    ])
```

In [ ]:

```
jittered
```

In [ ]:

```
jittered.scatter(0, 1, group = 'Class')
```

Distance

In [ ]:

```
def distance(pt1, pt2):
    """Return the distance between two points, represented as arrays"""
    return np.sqrt(sum((pt1 - pt2)**2))

def row_distance(row1, row2):
    """Return the distance between two numerical rows of a table"""
    return distance(np.array(row1), np.array(row2))
```

In [ ]:
```
attributes = patients.drop('Class')
attributes.show(3)
```

In [ ]:
```
row_distance(attributes.row(0), attributes.row(1))
```

In [ ]:
```
row_distance(attributes.row(0), attributes.row(2))
```

In [ ]:
```
row_distance(attributes.row(0), attributes.row(0))
```

Classification Procedure

In [ ]:
```
def distances(training, example):
    """Compute distance between example and every row in training.
    Return training augmented with Distance column"""
    distances = make_array()
    attributes = training.drop('Class')
    for row in attributes.rows:
        distances = np.append(distances, row_distance(row, example))
    return training.with_column('Distance', distances)
```

In [ ]:
```
patients.take(15)
```

In [ ]:
```
example = attributes.row(15)
example
```

In [ ]:
```
distances(patients.exclude(15), example).sort('Distance')
```

In [ ]:
```
def closest(training, example, k):
    """Return a table of the k closest neighbors to example"""
    return distances(training, example).sort('Distance').take(np.arange(k))
```

In [ ]:
```
closest(patients.exclude(15), example, 5)
```

In [ ]:
```
def majority_class(topk):
    """Return the class with the highest count"""
    return topk.group('Class').sort('count', descending=True).column(0).item(0)

def classify(training, example, k):
    "Return the majority class among the k nearest neighbors of example"
    return majority_class(closest(training, example, k))
```

In [ ]:
```
classify(patients.exclude(15), example, 5)
```

In [ ]:
```
patients.take(15)
```

In [ ]:
```
new_example = attributes.row(10)
closest(patients.exclude(10), example, 5)
```

In [ ]:
```
classify(patients.exclude(10), new_example, 5)
```

In [ ]:
```
patients.take(10)
```

Evaluation

In [ ]:
```
patients.num_rows
```

In [ ]:
```
shuffled = patients.sample(with_replacement=False) # Randomly permute the rows
training_set = shuffled.take(np.arange(342))
test_set  = shuffled.take(np.arange(342, 683))
```

In [ ]:
```
def evaluate_accuracy(training, test, k):
    """Return the proportion of correctly classified examples
    in the test set"""
```

```
    test_attributes = test.drop('Class')
    num_correct = 0
    for i in np.arange(test.num_rows):
        c = classify(training, test_attributes.row(i), k)
        num_correct = num_correct + (c == test.column('Class').item(i))
    return num_correct / test.num_rows
```

In [ ]:

```
evaluate_accuracy(training_set, test_set, 5)
```

In [ ]:

```
evaluate_accuracy(training_set, test_set, 3)
```

In [ ]:

```
evaluate_accuracy(training_set, test_set, 11)
```

In [ ]:

```
evaluate_accuracy(training_set, training_set, 1)
```

In [ ]: