In [1]:
```
%matplotlib inline
import matplotlib
import seaborn as sns
sns.set()
matplotlib.rcParams['figure.dpi'] = 144
```

K Nearest Neighbors

© The Data Incubator

In this notebook we'll explore the K Nearest Neighbors algorithm. As the name implies, the prediction for a new point is based on the closest k points in the training set (where closest is usually defined in terms of Euclidean distance on the p-dimensional feature space). In math terms, we're taking the Euclidean distance between the p-vectors $X_{j.}$ as our metric. As a classification algorithm, it takes a majority vote of the nearest k neighbors. As a regression algorithm, it takes the average label of the nearest k neighbors.

In [2]:
```
# Load and display a sample of the iris dataset

from sklearn import neighbors, datasets
import pandas as pd
import numpy as np

# Load the data
# The iris data set labels 3 flower types (y) by 4 different attributes of the
flower (X).
# For more about the attributes, see http://mldata.org/repository/data/viewslu
g/datasets-uci-iris/
data = datasets.load_iris()
X = pd.DataFrame(data.data, columns=["sepal_length", "sepal_width", "petal_leg
nth", "petal_width"])
y = pd.Series(data.target)
X.head()
```

Out[2]:

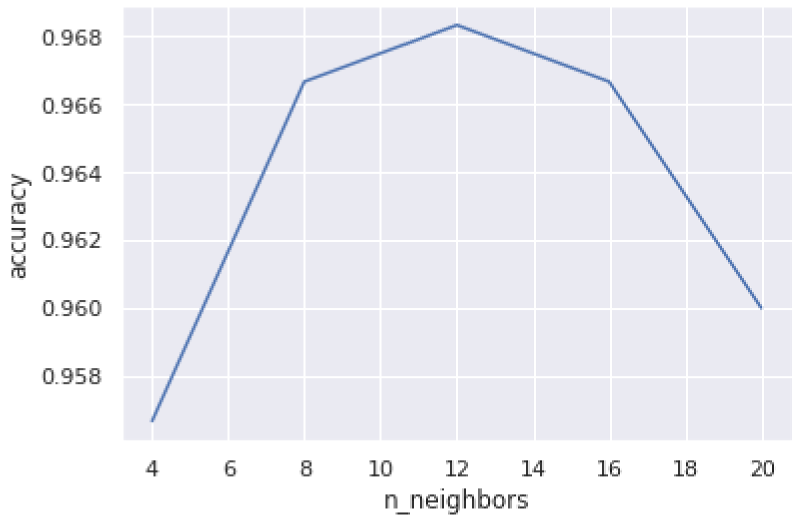
	sepal_length	sepal_width	petal_legnth	petal_width
0	5.1	3.5	1.4	0.2
1	4.9	3.0	1.4	0.2
2	4.7	3.2	1.3	0.2
3	4.6	3.1	1.5	0.2
4	5.0	3.6	1.4	0.2

In [3]:
```
# Process the nearest neighbors
nearest_neighbors = neighbors.NearestNeighbors(n_neighbors=5, algorithm='ball_
tree')
nearest_neighbors.fit(X)
distances, indices = nearest_neighbors.kneighbors(X)

# Column $k$ gives the distances between each point and its $k$-th nearest nei
ghbor from 0 to 4
# Column 0 is always itself.
# Each row is a separate point (note the first column)
print("Distance")
print(pd.DataFrame(distances).head())
print("Indices")
print(pd.DataFrame(indices).head())
```

```
Distance
     0         1         2         3         4
0  0.0  0.100000  0.141421  0.141421  0.141421
1  0.0  0.141421  0.141421  0.173205  0.173205
2  0.0  0.141421  0.244949  0.264575  0.264575
3  0.0  0.141421  0.173205  0.223607  0.244949
4  0.0  0.141421  0.173205  0.173205  0.223607
Indices
   0   1   2   3   4
0  0  17   4  39  27
1  1  45  12  34  37
2  2  47   3  12   6
3  3  47  29  30   2
4  4   0  17  40   7
```

Tuning k and other hyperparameters

In [4]:
```
# We illustrate the effect of the number of neigbhors used on accuracy.

from sklearn import neighbors, model_selection
import matplotlib.pylab as plt

cv = model_selection.ShuffleSplit(n_splits=20, test_size=0.2, random_state=42)
param_grid = { "n_neighbors": range(4, 24, 4) }
nearest_neighbors_cv = model_selection.GridSearchCV(neighbors.KNeighborsClassi
fier(),
                                                param_grid=param_grid, cv=cv,
                                                scoring='accuracy')

nearest_neighbors_cv.fit(X, y)
cv_results = nearest_neighbors_cv.cv_results_
cv_accuracy = pd.DataFrame.from_dict({'n_neighbors': [rec['n_neighbors'] for r
ec in cv_results['params']],
                                      'accuracy': cv_results['mean_test_score'
]})

plt.plot(cv_accuracy.n_neighbors, cv_accuracy.accuracy)
plt.xlabel('n_neighbors')
plt.ylabel('accuracy')
plt.show()
```



Questions:

1. Is this algorithm parametric on non-parametric?
2. As `n_neighbors` gets larger, the model becomes more accurate and then less. What's happening?
3. For the classification version, what happens when you have one class being over represented?
4. The notion of distance in nearest neighbors is Euclidean in feature space. For what kinds of input data might this be a problem?

Normalizing features

Based on your answer to the last question, we will want to scale the individual features so that they have similar variance. For each feature, we are going to subtract the mean and divide by the standard deviation:

$$X'_{ji} = \frac{X_{ji} - \mu_i}{\sigma_i}$$

where μ_i is the mean of the i-th column (or feature) and σ_i is the standard deviation.

Question: For k-Nearest-Neighbors, was subtracting the mean necessary? For what other algorithms might subtracting the mean help?

Exercise: Let's put all each feature of the data set on the same scale by normalizing.

In [5]:
```
from sklearn import preprocessing

# first, let's see what the standard deviations are.
scaler = preprocessing.StandardScaler(copy=True).fit(X)
pd.DataFrame({
    "Mean": scaler.mean_,
    "Std": scaler.scale_
}, index=X.columns)
```

Out[5]:

	Mean	Std
sepal_length	5.843333	0.825301
sepal_width	3.054000	0.432147
petal_legnth	3.758667	1.758529
petal_width	1.198667	0.760613

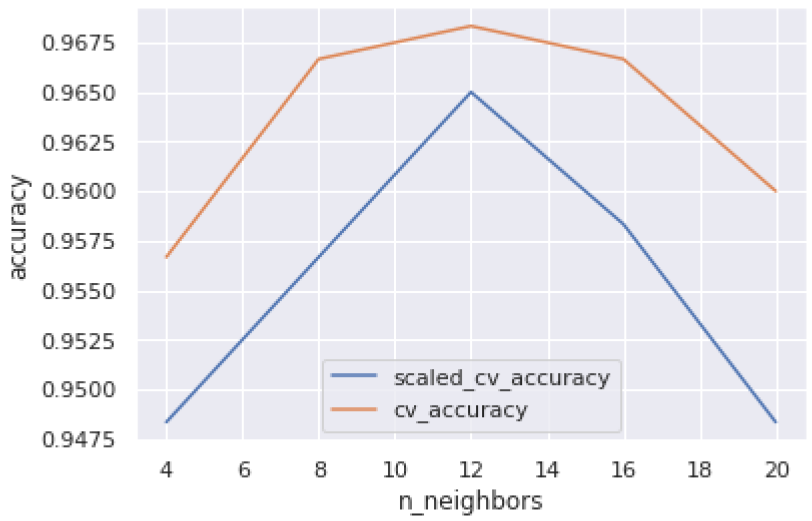
Exercise: It looks like some of the features are on a larger numerical scale. Let's put all the data on the same scale. This also subtracts the mean

In [6]:
```
X_copy = X.copy()  # Feature (bug?): transform modifies the original
X_scaled = scaler.transform(X_copy)
pd.DataFrame({
    "Mean": X_scaled.mean(axis=0),
    "Std": X_scaled.std(axis=0)
})
```

Out[6]:

	Mean	Std
0	-1.468455e-15	1.0
1	-1.657933e-15	1.0
2	-1.515825e-15	1.0
3	-8.052818e-16	1.0

In [7]:
```
# We are going to first scale and then apply KNN.
# Pipeline allows us to chain together multiple transformers and then a regres
sor or classifier

from sklearn.pipeline import Pipeline

scaled_nearest_neighbors = Pipeline([('scaling', preprocessing.StandardScaler(
copy=True)),
                                     ('neighbors', neighbors.KNeighborsClassif
ier())])

param_grid = {"neighbors__n_neighbors": range(4, 24, 4)}    # parameters to Pi
peline take the form [label]__[estimator_param]
scaled_nearest_neighbors_cv = model_selection.GridSearchCV(scaled_nearest_neig
hbors,
                                                       param_grid=param_grid,
cv=cv,
                                                       scoring='accuracy')

scaled_nearest_neighbors_cv.fit(X, y)
scaled_cv_results = scaled_nearest_neighbors_cv.cv_results_
scaled_cv_accuracy = pd.DataFrame.from_dict(
    {'n_neighbors': [rec['neighbors__n_neighbors'] for rec in scaled_cv_result
s['params']],
    'accuracy': scaled_cv_results['mean_test_score']})

plt.plot(scaled_cv_accuracy.n_neighbors, scaled_cv_accuracy.accuracy, label="s
caled_cv_accuracy")
plt.plot(cv_accuracy.n_neighbors, cv_accuracy.accuracy, label="cv_accuracy")
plt.xlabel('n_neighbors')
plt.ylabel('accuracy')
plt.legend(loc='lower center')
plt.show()
```



Additional resources

1. Wikipedia (http://en.wikipedia.org/wiki/K-nearest_neighbors_algorithm)
2. The Scikit Learn package implements K Nearest Neighbors (http://scikit-learn.org/stable/modules/neighbors.html). It has both a `KNeighborsClassifier` (http://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html) and `KNeighborsRegressor` (http://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsRegressor.html).

Exit Tickets

1. In KNN, does most computation take place during training or during testing?
2. After training, what must the model retain in memory in order to make predictions?
3. What are algorithms like `KD-Ball` and `KD-Tree` meant to do? And what's their drawback?

Spoilers

Answers

Tuning k and hyperparameters

1. The algorithm is non-parametric.
2. As `n_neighbors` gets larger, we're increasing variance and decreasing bias so we expect to see greater and than less accuracy.
3. For the classification version, if one class is over-represented, then it will always win a majority vote. Using an average of the nearest k neighbors as a predictor may be useful.
4. The notion of distance in nearest neighbors is Euclidean in feature space. This is really bad if the features have very different scales.

Normalizing features

1. It is probably good to normalize the data by dividing all the features by their standard deviation. Note that subtracting a mean is not necessary.
2. There are three general cases where you will want to scale your features. The first is when you are using Euclidean distance measures (e.g. k-means and k-nearest neighbors). The second is when you are optimizing your model using gradient descent/ascent (e.g. linear regression, logistic regression, SVMs, neural networks), where the weights associated with large scale features will update faster than those associated with small scale features. The third is when you want to perform something like a PCA and find the "direction" of maximum variance in your data. The large scale features will have a greater influence on the direction of your principal components. Decision trees, on the other hand, are scale-invariant.

Exit Tickets

1. Testing. Training is just storing all of the data. In testing, we have to look up which training data is nearest to the testing data.
2. The full X and y of the training set.
3. `KD-Trees` divide space into recursive partitions. (Quad tree in 2D, oct tree in 3D, etc.) This structure can be used to identify neighbors more efficiently, at least in small (<20) dimensions. A ball tree partitions space into set of nested hyperspheres. Construction is costly, but lookup is reasonably fast, even in large D.

Copyright © 2019 The Data Incubator. All rights reserved.