QBUS6850: Tutorial 5 - Scalable Classification Methods

Objectives

- Get familiar with kNN and K-means algorithm
- Apply kNN and K-means algorithm by using scikit-learn library

Tasks

- kNN Classification
- kNN Classification - Practical Example and Optimising k
- K-means Clustering

1. kNN Classification

k Nearest Neighbors is a very simple, yet highly effective and scalable classifcation method. It is supervised in the sense that we build a decision mesh based on observed training data. It is non-parametric, meaning it makes no assumptions about the structure or form of the generative function. It is also instance-based which means that it doesn't explicitiy learn a model. Instead it simply uses the training data as "knowledge" about the domain. Practically this means that the training data is only used when a specific query is made ("which data points am I closest to?").

This highly flexible and "just in time" nature means that it scales well to hugre volumes of data and is often the first port of call for classifying massive data sets.

Since kNN is non-parametric it is generally considered to have high variance and low bias. However we can tune this by increasing k - the number of neighbours to average over. A higher value for k leads to a smoother decision mesh/boundary and lower variance. For each application you must tune k on your training data for optimal performance.

Step1: kNN can perform poorly if your features are not all at the same scale. Therefore it is good to normalise them to be within the same range, otherwise features with larger or smaller scales can dominate.

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap
from sklearn import neighbors
from sklearn.datasets import make_blobs

X, y = make_blobs(100, \
                  centers= [[2, 2],[1, -2]], \
                  cluster_std=[1.5, 1.5], \
                  random_state = 1)
```

Step2: We will generate a set of x,y locations. These will then be fed into the predict() function of our KNN classifiers. Then we can plot the result of each x,y position to show the boundary of the classifiers.

```
"""
Initialise some plotting variables
"""
h = .02  # step size in the mesh

# Create color maps
cmap_light = ListedColormap(['#FFAAAA', '#AAAAFF'])
cmap_bold = ListedColormap(['#FF0000', '#0000FF'])

# Plot the decision boundary. For that, we will assign a color to each
# point in the mesh [x_min, m_max]x[y_min, y_max].
x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
                     np.arange(y_min, y_max, h))
```

Step3: Do KNN Classification for k = 1

```
n_neighbor_1 = 1

knn_clf_1 = neighbors.KNeighborsClassifier(n_neighbor_1, weights='uniform' )
knn_clf_1.fit(X, y)

Z_1 = knn_clf_1.predict(np.c_[xx.ravel(), yy.ravel()])
# np.c_ is a way to combine 1D arrays into 2D array
# Put the result into a color plot
Z_1 = Z_1.reshape(xx.shape)
```

Step4: Do KNN Classification for k = 50

```
n_neighbor_2 = 50

knn_clf_50 = neighbors.KNeighborsClassifier(n_neighbor_2, weights='uniform' )
knn_clf_50.fit(X, y)

Z_50 = knn_clf_50.predict(np.c_[xx.ravel(), yy.ravel()])

Z_50 = Z_50.reshape(xx.shape)
```

Step5: Plot the decision boundary and the classification of indivudal data points

```
fig5 = plt.figure(figsize=(10,5))

plt.subplot(1, 2, 1)
plt.gcf().set_size_inches(10, 5)
plt.pcolormesh(xx, yy, Z_1, cmap=cmap_light)

# Plot also the training points
plt.scatter(X[:, 0], X[:, 1], c=y, cmap=cmap_bold)
plt.xlim(xx.min(), xx.max())
plt.ylim(yy.min(), yy.max())
plt.title("2-Class classification (k = {0})".format(n_neighbor_1))

plt.subplot(1, 2, 2)
plt.gcf().set_size_inches(10, 5)
plt.pcolormesh(xx, yy, Z_50, cmap=cmap_light)
```

```
plt.scatter(X[:, 0], X[:, 1], c=y, cmap=cmap_bold)
plt.xlim(xx.min(), xx.max())
plt.ylim(yy.min(), yy.max())
plt.title("2-Class classification (k = {0})".format(n_neighbor_2))
```

Observations

Note that with small k we have high variance/low bias (model fits closely to training data) and with larger k we have low variance/high bias (model fits more closely to generative data process).

In other words it is likely that a small k value will lead to overfitting, while larger k can lead to underfitting.

2. kNN Classification - Practical Example and Optimising k

We can use kNN Classification for all sorts of data. Here we will use kNN to classify images of hand written English characters.

In this dataset there are 1797 character image. Each image is 8x8 pixels i.e. of dimension 64. We reshape each image to a 64x1 vector. Therefore we are "searching" in 64 dimensional space.

We will use cross validation on the training set to determine performance of a range of k values and then perform final evaluation on the test set.

The dataset is built into scikit. However you will need to download it from the web as it is too large to be bundled directly.

Step1: Load the dataset by using load_digits() function

```
from sklearn import datasets
from sklearn.cross_validation import train_test_split
from sklearn.metrics import classification_report, confusion_matrix
from sklearn.model_selection import cross_val_score

digits = datasets.load_digits()
```

Step2: Let's take a look at some sample images. The images in this dataset come from postal codes so they are all numeric.

```
fig6 = plt.figure()
plt.gray()
for i in range(1, 11):
    plt.subplot(1, 11, i)
    plt.imshow(digits.images[i - 1])
fig6
```

Step3: Split the data into train, validation and test sets

```
trainData, testData, trainLabels, testLabels = train_test_split(np.array(digits.data), \
                                                                 digits.target, \
                                                                 test_size=0.25, \
                                                                 random_state=42)

print("m Training Points: {}".format(len(trainLabels)))
print("m Test points: {}".format(len(testLabels)))
```

Optimising k

Step4: To determine the optimal k we will train our classifier on the training data with various vlaues of k. We will then evaluate the performance of each model on the validation set.

```
# Store cv score for each k
cv_scores = []
k_vals = []

for k in range(1, 30, 2):
    model = neighbors.KNeighborsClassifier(n_neighbors=k)
    scores = cross_val_score(model, \
                             trainData, \
                             trainLabels, \
                             cv=10, \
                             scoring='accuracy')
    score = scores.mean()
    print("k={0}, cv_score={1:.2f}".format(k, score * 100))
    cv_scores.append(score)
    k_vals.append(k)

# Find best performing k
idx = np.argmax(cv_scores)
print("k={0} achieved highest accuracy of {1:.2f}"\
      .format(k_vals[idx], cv_scores[idx] * 100))
```

Step5: We now know that the best k for this set of training and validation data is k = 1. We may find a different result if we reshuffle the data or obtain new data. Finally we can build our optimal classifier and evaluate on the test set

```
model = neighbors.KNeighborsClassifier(n_neighbors = k_vals[idx])
model.fit(trainData, trainLabels)
predictions = model.predict(testData)

# Final classification report
print(classification_report(testLabels, predictions))
print(confusion_matrix(testLabels, predictions))
```

Observations

Suprisingly our kNN classifier achieves near perfect classification accuracy on this data set.

Remember: more complex models do not guarantee better results. Sometimes it is better to stick with simple and effective methods.

Note that this is a multi-class problem. The details of multi-class classification and its confusion matrix will be explained in later lecture.

3. K-means clustering

K-means is a very simple clustering method. It performs suprisingly well in practice and its simplicity means that it can be applied to massive datasets with ease.

K-means follows this basic outline:

1. Initialise centtroids of clusters
2. Assign labels of data according to centroids
3. Produce new centroids from current clusters
4. Repeat 2-3 until convergence (no change in cluster assignments)

Weaknesses

K-means is unsupervised and non-parametric. It does not explicity prescribe a model. However its avergaing method performs best under certain conditions, such as Gaussian distributed clusters. It is also affected greatly by choice of initial cluster centroids.

We will illustrate the following weaknesses of k-means:

- Sensitivity to number of clusters. You must know before hand the true number of clusters in your data, otherwise it will perform poorly
- Assumption of Gaussian distributed data. If data is anisotrpoically distributed (stretched) then k-means often fails.
- Clusters should have equal variance
- Clusters should have equal size in space

Step1: Lets generate some synthetic data with three clusters

```
import numpy as np
import matplotlib.cm as cm
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans

n_samples = 1500
random_state = 170
X, y = make_blobs(n_samples=n_samples, random_state=random_state)
```

Incorrect number of clusters

Step2: If we incorrectly guess the number of clusters K-means will produce a defective result

```
y_pred = KMeans(n_clusters=2, random_state=random_state).fit_predict(X)

fig1 = plt.figure(figsize=(12, 6))

plt.subplot(1,2,1)
plt.scatter(X[:, 0], X[:, 1], \
            c=[cm.spectral(float(i) /10) for i in y])
plt.title("Ground Truth - Incorrect Number of Clusters")

plt.subplot(1,2,2)
plt.scatter(X[:, 0], X[:, 1], \
            c=[cm.spectral(float(i) /10) for i in y_pred])
plt.title("K-means Clusters - Incorrect Number of Clusters")
```

Anisotropicly distributed data

Step3: If data is anisotrpoically distributed (stretched) instead of Gaussian distributed then k-means often fails.

```
transformation = [[ 0.60834549, -0.63667341], [-0.40887718, 0.85253229]]
X_aniso = np.dot(X, transformation)
y_pred = KMeans(n_clusters=3, random_state=random_state).fit_predict(X_aniso)

fig2 = plt.figure(figsize=(12, 6))

plt.subplot(1,2,1)
plt.scatter(X_aniso[:, 0], X_aniso[:, 1], \
            c=[cm.spectral(float(i) /10) for i in y])
plt.title("Ground Truth - Anisotropicly Distributed Clusters")

plt.subplot(1,2,2)
plt.scatter(X_aniso[:, 0], X_aniso[:, 1], \
            c=[cm.spectral(float(i) /10) for i in y_pred])
plt.title("K-means Clusters - Anisotropicly Distributed Clusters")
```

Step4: Clusters with unequal variance

```
X_varied, y_varied = make_blobs(n_samples=n_samples,
                                cluster_std=[1.0, 2.5, 0.5],
                                random_state=random_state)
y_pred = KMeans(n_clusters=3, random_state=random_state).fit_predict(X_varied)

fig3 = plt.figure(figsize=(12, 6))

plt.subplot(1,2,1)
plt.scatter(X_varied[:, 0], X_varied[:, 1], \
            c=[cm.spectral(float(i) /10) for i in y_varied])
plt.title("Ground Truth - Unequal Variance")

plt.subplot(1,2,2)
plt.scatter(X_varied[:, 0], X_varied[:, 1], \
            c=[cm.spectral(float(i) /10) for i in y_pred])
plt.title("K-means Clusters - Unequal Variance")
```

Step5: Clusters with unequal size

```
X_filtered = np.vstack((X[y == 0][:500], X[y == 1][:100], X[y == 2][:7]))
y_filtered = np.vstack((np.zeros((500,1)), np.ones((100,1)), 2*np.ones((7,1))))

y_pred = KMeans(n_clusters=3, random_state=random_state).fit_predict(X_filtered)

fig4 = plt.figure(figsize=(12, 6))

plt.subplot(1,2,1)
plt.scatter(X_filtered[:, 0], X_filtered[:, 1], \
            c=[cm.spectral(float(i) /10) for i in y_filtered])
plt.title("Ground Truth - Unevenly Sized Clusters")

plt.subplot(1,2,2)
plt.scatter(X_filtered[:, 0], X_filtered[:, 1], \
            c=[cm.spectral(float(i) /10) for i in y_pred])
plt.title("K-means Clusters - Unevenly Sized Clusters")
```