K-Nearest Neighbor

Data Mining for Business Analytics in Python

Shmueli, Bruce, Gedeck & Patel

Characteristics of K-NN

- Data-driven, not model-driven

  (Relies primarily on the data itself to draw conclusions, make predictions, or inform decisions, not the use of mathematical or statistical models)

- Makes no assumptions about the data

- Simple idea: Classify a record like similar records

- Classification and Regression

Basic Concept

- For a given record to be classified, identify nearby records
- "Near" means records with similar predictor values $X_1, X_2, \ldots X_p$
- Classify the record as whatever the predominant class is among the nearby records (the "neighbors")

Import Needed Functionality

```
import pandas as pd
from sklearn import preprocessing
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from sklearn.neighbors import NearestNeighbors,
   KNeighborsClassifier
import matplotlib.pylab as plt
```

How to measure "nearby"?

The most popular distance measure is **Euclidean distance.**
The Euclidean distance between two records (x_1, x_2, ..., x_p) and (u_1, u_2, ..., u_p) in a p-dimensional space is:

$$\sqrt{(x_1 - u_1)^2 + (x_2 - u_2)^2 + \cdots + \left(x_p - u_p\right)^2}$$

- Typically, predictor variables are first normalized (=standardized) to put them on comparable scales.
- Otherwise, metrics with large scales dominate.

Converting Categorical Variables to Binary Dummies

It does not make sense to calculate Euclidean distance between two non-numeric categories (e.g., cookbooks and maps, in a bookstore). Therefore, before k-NN can be applied, categorical variables must be converted to binary dummies. In contrast to the situation with statistical models such as regression, all m binaries should be created and used with k-NN. While mathematically this is redundant, since $m - 1$ dummies contain the same information as m dummies, this redundant information does not create the multicollinearity problems that it does for linear models. Moreover, in k-NN the use of $m - 1$ dummies can yield different classifications than the use of m dummies, and lead to an imbalance in the contribution of the different categories to the model.

Choosing k

K is the number of nearby neighbors to be used to classify the new record

K=1 means use the single nearest record
K=5 means use the 5 nearest records

Typically choose that value of *k* which has lowest error rate in validation data

Low *k* vs. High *k*

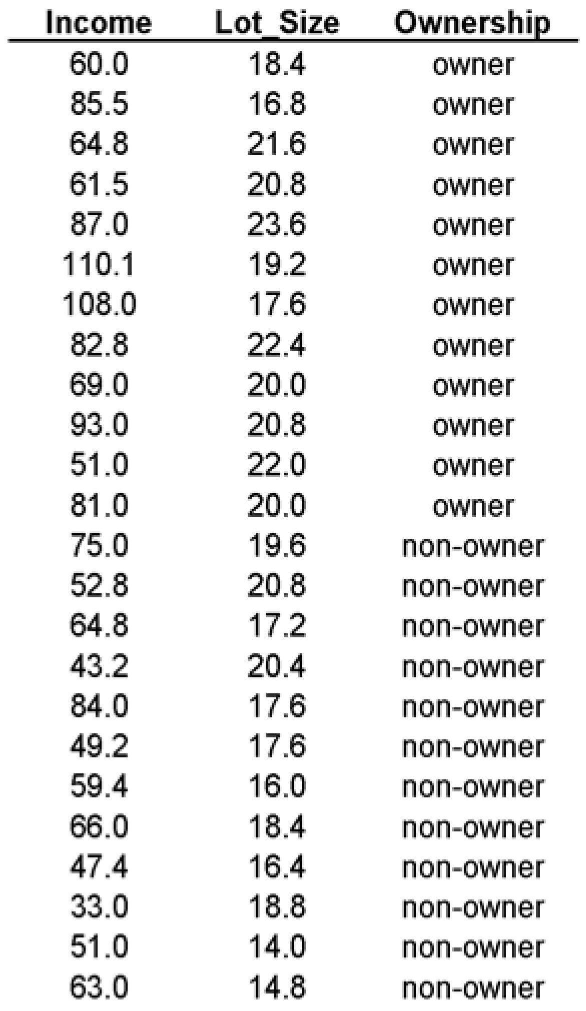
- Low values of *k* (1, 3, …) capture local structure in data (but also noise)
- High values of *k* provide more smoothing, less noise, but may miss local structure

Note: The extreme case of k = n (i.e., the entire data set) is the same as the "naïve rule" (classify all records according to majority class)

Example: Riding Mowers

Data: 24 households classified as owning or not owning riding mowers

Predictors: Income, Lot Size



Income	Lot_Size	Ownership
60.0	18.4	owner
85.5	16.8	owner
64.8	21.6	owner
61.5	20.8	owner
87.0	23.6	owner
110.1	19.2	owner
108.0	17.6	owner
82.8	22.4	owner
69.0	20.0	owner
93.0	20.8	owner
51.0	22.0	owner
81.0	20.0	owner
75.0	19.6	non-owner
52.8	20.8	non-owner
64.8	17.2	non-owner
43.2	20.4	non-owner
84.0	17.6	non-owner
49.2	17.6	non-owner
59.4	16.0	non-owner
66.0	18.4	non-owner
47.4	16.4	non-owner
33.0	18.8	non-owner
51.0	14.0	non-owner
63.0	14.8	non-owner

Normalizing data, based on training set

```
mower_df = pd.read_csv('RidingMowers.csv')
trainData, validData = train_test_split(mower_df, test_size=0.4, random_state=26)

# use the training data to learn the transformation.
scaler = preprocessing.StandardScaler()
scaler.fit(trainData[['Income', 'Lot_Size']]) # Note use of array of column names
# Calculating "scaler.fit" means that you are calculating the mean and standard deviation of these columns.

# Transform the full dataset
mowerNorm = pd.concat([pd.DataFrame(scaler.transform
    (mower_df[['Income', 'Lot_Size']]),columns=['zIncome',
    'zLot_Size']), mower_df[['Ownership', 'Number']]], axis=1)
trainNorm = mowerNorm.iloc[trainData.index]
validNorm = mowerNorm.iloc[validData.index]
```

Consider a new record to be classified

New household with \$60,000 income and lot size 20,000 ft^2

```
newHousehold = pd.DataFrame([{'Income': 60, 'Lot_Size': 20}])

newHouseholdNorm = pd.DataFrame(scaler.transform(newHousehold),
    columns=['zIncome', 'zLot_Size’])
```

Find Nearest Neighbors

```
# use NearestNeighbors from scikit-learn to compute knn

from sklearn.neighbors import NearestNeighbors

knn = NearestNeighbors(n_neighbors=3)
knn.fit(trainNorm.iloc[:, 0:2])
distances, indices = knn.kneighbors(newHouseholdNorm)
```

* When you execute knn.fit(trainNorm.iloc[:, 0:2]), you are fitting a k-Nearest Neighbors (KNN) model to your training data. Specifically, you are using the training data to create a KNN model that can be used for making predictions or classifications.

Results

```
# indices is a list of lists, we are only
# interested in the first element, which is
# the 3 neighbors for the new record

trainNorm.iloc[indices[0], :]

Output
        zIncome  zLot_Size Ownership
3     -0.409776  0.743358    Owner
13    -0.804953  0.743358    Nonowner
0     -0.477910 -0.174908    Owner
```

Classifier Training
Accuracy for Different *k*

(measured on validation data)

```
train_X = trainNorm[['zIncome', 'zLot_Size']]
train_y = trainNorm['Ownership']
valid_X = validNorm[['zIncome', 'zLot_Size']]
valid_y = validNorm['Ownership']

# Train a classifier for different values of k
results = []
for k in range(1, 15):
   knn = KNeighborsClassifier(n_neighbors=k).fit(train_X, train_y)
    results.append({
       'k': k,
       'accuracy': accuracy_score(valid_y, knn.predict(valid_X))
})

output = pd.DataFrame(results)
print(output)
```

Accuracy is the ratio of the number of correct predictions to the total number of predictions made. It is defined as:
Accuracy = Number of correct predictions / Total number of predictions

Accuracy for Different *k*

(measured on validation data)

```
      k    accuracy
0     1    0.6
1     2    0.7
2     3    0.8
3     4    0.9  <==      maximum accuracy with smallest k
4     5    0.7
5     6    0.9
6     7    0.9
7     8    0.9
8     9    0.9
9    10    0.8
10   11    0.8
11   12    0.9
12   13    0.4
13   14    0.4
```

Note: for even k, ties are broken randomly

Using K-NN for Prediction
(for Numerical Outcome)

- Instead of “majority vote determines class” use **average** of response values.
- May be a weighted average, weight decreasing with distance.
- In scikit-learn, we can use KNeighborsRegressor to compute k-NN numerical predictions for the validation set.
- Another modification is in the error metric used for determining the “best k.” Rather than the overall error rate used in classification, RMSE (root-mean-squared error) or another prediction error metric should be used in prediction.

Class Imbalance

Some strategies to handle class imbalance:

1. **Weighted Distance Metrics:** You can assign different weights to the distances when calculating the nearest neighbors. For example, you can use an inverse distance weighting, where nearer neighbors have a higher influence, and distant neighbors have a lower influence. This approach can help mitigate the bias caused by class imbalance.
2. **Over-sampling and Under-sampling:** Over-sampling the minority class and under-sampling the majority class can be used to balance the class distribution. Techniques like Synthetic Minority Over-sampling Technique (SMOTE) can help generate synthetic examples for the minority class. However, these techniques should be applied cautiously to avoid overfitting.
3. **Cross-Validation:** When evaluating your KNN model, consider using techniques like stratified k-fold cross-validation. This ensures that each fold maintains the same class distribution as the entire dataset. It helps in obtaining more robust and unbiased performance estimates.
4. **Selecting the Appropriate Value of K:** The choice of the value of k (the number of neighbors) can impact the model's sensitivity to class imbalance. Smaller values of k may make the model more sensitive to noise, while larger values may make it more biased toward the majority class. Experiment with different values of k to find the best balance.
5. **Ensemble Methods:** Consider using ensemble techniques like Balanced Random Forests or Easy Ensemble, which combine multiple models to mitigate the effects of class imbalance.
6. **Evaluation Metrics:** When assessing the model's performance, focus on evaluation metrics beyond accuracy, such as precision, recall, F1-score, or the area under the receiver operating characteristic (ROC-AUC). These metrics provide a more comprehensive view of how well the model handles class imbalance.
7. **Feature Engineering:** Feature selection and engineering can be used to improve the model's sensitivity to the minority class. Selecting relevant features and transforming them can have a positive impact on the model's performance.
8. **Collect More Data:** If feasible, collecting more data for the minority class can help improve the model's performance and reduce class imbalance.

Using K-NN for Prediction
(for Numerical Outcome) - Example

```
from sklearn.neighbors import KNeighborsRegressor

knn_reg = KNeighborsRegressor(n_neighbors=k)

knn_reg.fit(train_X, train_y)

y_pred = knn_reg.predict(new record)
```

Advantages

- Simple
- No assumptions required about Normal distribution, etc.
- Effective at capturing complex interactions among variables without having to define a statistical model

Shortcomings

- Required size of training set increases exponentially with # of predictors, *p*

  This is because expected distance to nearest neighbor increases with *p* (with large vector of predictors, all records end up “far away” from each other)

- In a large training set, it takes a long time to find distances to all the neighbors and then identify the nearest one(s)
- These constitute “curse of dimensionality”

Dealing with the Curse

- Reduce dimension of predictors (e.g., with PCA)
- Computational shortcuts that settle for "almost nearest neighbors"

Pandora`s Story

Pandora is an Internet music radio service that allows users to build customized "stations" that play music similar to a song or artist that they have specified. Pandora uses a k-NN style clustering/classification process called the Music Genome Project to locate new songs or artists that are close to the user-specified song or artist.

Pandora was the brainchild of Tim Westergren, who worked as a musician and a nanny when he graduated from Stanford in the 1980s. Together with Nolan Gasser, who was studying medieval music, he developed a "matching engine" by entering data about a song's characteristics into a spreadsheet. The first result was surprising—a Beatles song matched to a Bee Gees song, but they built a company around the concept. The early days were hard—Westergren racked up over $300,000 in personal debt, maxed out 11 credit cards, and ended up in the hospital once due to stress-induced heart palpitations. A venture capitalist finally invested funds in 2004 to rescue the firm, and as of 2013, it is listed on the NY Stock Exchange. In simplified terms, the process works roughly as follows for songs:

Pandora has established hundreds of variables on which a song can be measured on a scale from 0 to 5. Four such variables from the beginning of the list are: (1) Acid Rock Qualities, (2) Accordion Playing, (3) Acousti-Lectric Sonority, (4) Acousti-Synthetic Sonority.

Pandora pays musicians to analyze tens of thousands of songs, and rate each song on each of these attributes. Each song will then be represented by a row vector of values between 0 and 5, for example, for Led Zeppelin's Kashmir:

Kashmir 4 0 3 3 … (high on acid rock attributes, no accordion, etc.)

This step represents a costly investment, and lies at the heart of Pandora's value because these variables have been tested and selected as they accurately reflect the essence of a song, and provide a basis for defining highly individualized preferences.

The online user specifies a song that s/he likes (the song must be in Pandora's database).

Pandora then calculates the statistical distance between the user's song and the songs in its database. It selects a song that is close to the user-specified song and plays it.

The user then has the option of saying "I like this song," "I don't like this song," or saying nothing.

If "like" is chosen, the original song, plus the new song are merged into a 2-song cluster that is represented by a single vector comprising means of the variables in the original two song vectors.

If "dislike" is chosen, the vector of the song that is not liked is stored for future reference. (If the user does not express an opinion about the song, in our simplified example here, the new song is not used for further comparisons.)

Pandora looks in its database for a new song, one whose statistical distance is close to the "like" song cluster, and not too close to the "dislike" song. Depending on the user's reaction, this new song might be added to the "like" cluster or "dislike" cluster.

Over time, Pandora develops the ability to deliver songs that match a particular taste of a particular user. A single user might build up multiple stations around different song clusters. Clearly, this is a less limiting approach than selecting music in terms of which "genre" it belongs to.

While the process described above is a bit more complex than the basic "classification of new data" process described in this chapter, the fundamental process—classifying a record according to its proximity to other records—is the same at its core. Note the role of domain knowledge in this machine learning process—the variables have been tested and selected by the project leaders, and the measurements have been made by human experts.

Further reading: See www.pandora.com, Wikipedia's article on the Music Genome Project, and Joyce John's article "Pandora and the Music Genome Project," Scientific Computing, vol. 23, no. 10: 14, p. 40–41, Sep. 2006.