About K nearest neighbor (KNN)

- Unlike the greedy algorithm for decision tree models, KNN is a lazy algorithm.
- Knn is a simple model with a closed form bias-variance trade-off expression (see Wikipedia).
- Knn is completely different from Kmeans, though both of them need choose k 'something' for better accuracy. Knn is a supervised algorithm, while Kmeans is a unsupervised algorithm.

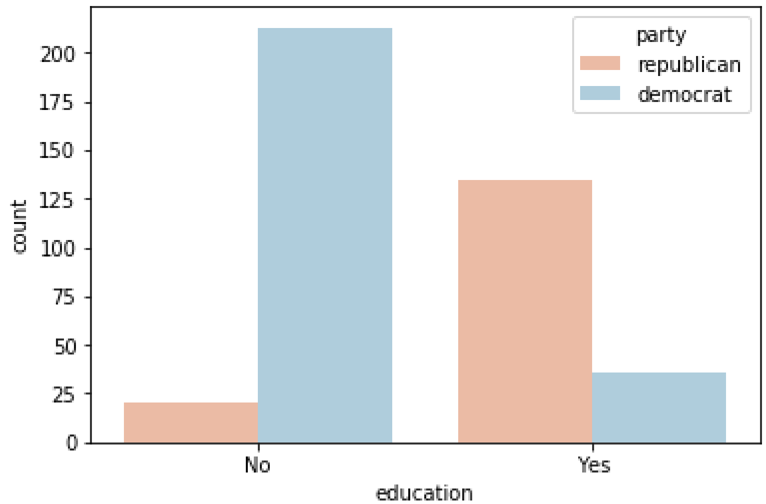
Exercise 1 -- Predict party affiliation from votes.

Numerical exploratory data analysis (EDA) and Visual EDA

From the countplot below, it seems like Democrats voted resoundingly against this bill, compared to Republicans. It is possible to simultaneously use multiple plots to show the relations of all of them very quickly.

In [5]:
```python
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np
import seaborn as sns
from sklearn.preprocessing import Imputer

filePath = "C:/Users/ljyan/Desktop/courseNotes/dataScience/machineLearning/data/"
filename = "house-votes-84.csv"
file = filePath+filename
df = pd.read_csv(file, sep =  ',', header = None)


# Or we can use: r transform normal string to raw string.
# df = pd.read_csv(r"C:\Users\ljyan\Desktop\courseNotes\dataScience\machineLearning\data\house-votes-84.csv",
#                    sep =  ',', header = None)

df.columns = ['party', 'infants', 'water', 'budget', 'physician', 'salvador','religious',
              'satellite', 'aid', 'missile', 'immigration', 'synfuels','education', 'superfund', 'crime', 'duty_
df[df == 'n'] = 0
df[df == 'y'] = 1
df[df == '?'] = np.nan

y = df['party'].values
X = df.drop('party', axis=1).values

imp = Imputer(missing_values= 'NaN', strategy= 'most_frequent', axis=0) #'most_frequent', 'mean' etc.
X=imp.fit_transform(X)
#If just writing imp.fit_transform(X), then the 'NaN' in X will not be replaced.

plt.figure() #Start with this sentence with new graph. Otherwise they will be overlayed
sns.countplot(x='education', hue='party', data=df, palette='RdBu')
plt.xticks([0,1], ['No', 'Yes'])
plt.show()
```

```
C:\Users\ljyan\Anaconda3\lib\site-packages\sklearn\utils\deprecation.py:58: DeprecationWarning: Class Imputer i
s deprecated; Imputer was deprecated in version 0.20 and will be removed in 0.22. Import impute.SimpleImputer f
rom sklearn instead.
  warnings.warn(msg, category=DeprecationWarning)
```



Predicting a single sample

In [11]:
```
from sklearn.neighbors import KNeighborsClassifier

knn = KNeighborsClassifier(n_neighbors = 6)
knn.fit(X,y)
y_pred = knn.predict(X)

#Prepare a new data for prediction
X_new = X[76]
print(X_new.shape) # Shape is (16,) not suitable for .predict in sklearn
X_new = X_new.reshape(-1,1) # shape is (16, 1) still not suitable for .predict()
X_new = X_new.reshape(1,-1) # shape is (1, 16). This is what we want. Is this suggesting scikit-learn use row ve
print(X_new.shape)

new_prediction = knn.predict(X_new) #single prediction. This is possible, although normally we predict a lot in 
print("Prediction: {}".format(new_prediction))
```

```
(16,)
(1, 16)
Prediction: ['democrat']
```

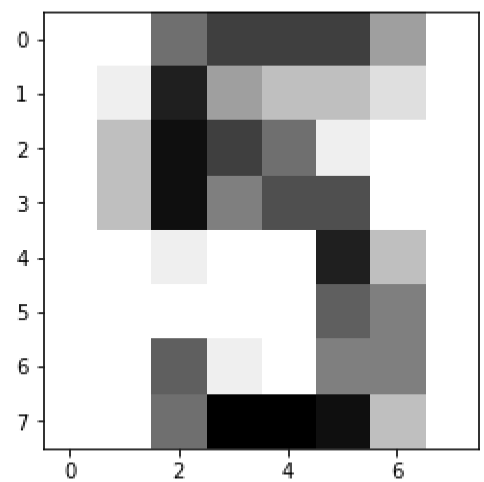
Exercise 2 -- Multi-class classification for the digits recognition

dataset

- MNIST digits recognition dataset, which has 10 classes, the digits 0 through 9.
- Reduced version of 8x8 image representing a handwritten digit. Each pixel is represented by an integer in the range 0 to 16, indicating varying levels of black.

In [12]:
```
from sklearn import datasets
import matplotlib.pyplot as plt

digits = datasets.load_digits() #There are also .load_iris() to load the iris data.
print(digits.keys()) # scikit-learn's built-in datasets are of type Bunch, which are dictionary-like objects. Thi

print(digits.images.shape)
print(digits.data.shape)

plt.imshow(digits.images[1010], cmap=plt.cm.gray_r, interpolation='nearest')
plt.show()
```

```
dict_keys(['data', 'target', 'target_names', 'images', 'DESCR'])
(1797, 8, 8)
(1797, 64)
```



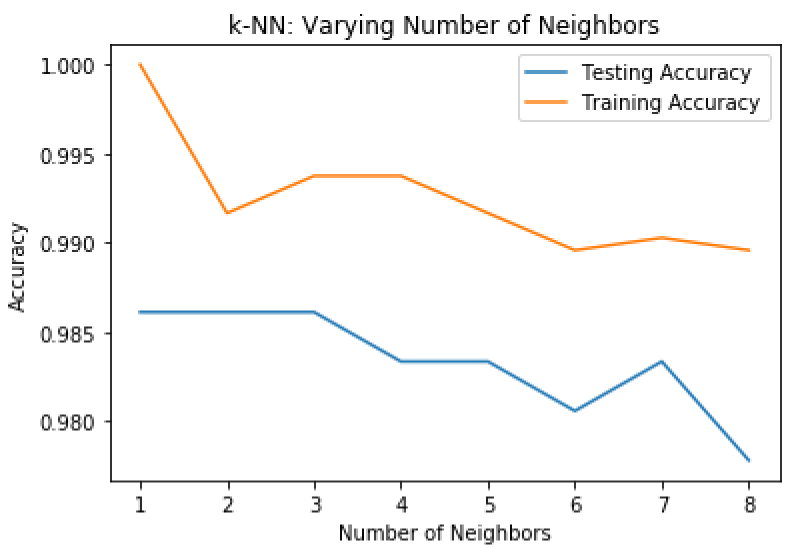
In [13]:
```python
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import train_test_split

X = digits.data
y = digits.target

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, random_state=42, stratify=y)
# This stratify parameter makes a split so that the proportion of values in the sample produced will be the same
# as the proportion of values provided to parameter stratify. For example, if variable y is a binary categorical
#variable with values 0 and 1 and there are 25% of zeros and 75% of ones, stratify=y will make sure that your rar
#split has 25% of 0's and 75% of 1's.

knn = KNeighborsClassifier(n_neighbors = 7)

knn.fit(X_train,y_train)
print(knn.score(X_test, y_test))
print(X.shape, X_train.shape, X_test.shape)
```

```
0.9833333333333333
(1797, 64) (1437, 64) (360, 64)
```

Overfitting and underfitting

In [14]:

```
import numpy as np
# Setup arrays to store train and test accuracies
neighbors = np.arange(1, 9)
train_accuracy = np.empty(len(neighbors))
test_accuracy = np.empty(len(neighbors))

# Loop over different values of k
for i, k in enumerate(neighbors):
    knn = KNeighborsClassifier(n_neighbors = k)
    knn.fit(X_train,y_train)
    train_accuracy[i] = knn.score(X_train,y_train)
    test_accuracy[i] = knn.score(X_test,y_test)

# Generate plot
plt.title('k-NN: Varying Number of Neighbors')
plt.plot(neighbors, test_accuracy, label = 'Testing Accuracy')
plt.plot(neighbors, train_accuracy, label = 'Training Accuracy')
plt.legend()
plt.xlabel('Number of Neighbors')
plt.ylabel('Accuracy')
plt.show()
```



It looks like the test accuracy is highest when using 3 and 5 neighbors. Using 8 neighbors or more seems to result in a simple model that underfits the data.