Supervised Learning with scikit-learn

July 26, 2017

1 Classification

1.1 Exploratory Data Analysis (video)

```
In [296]: %matplotlib inline
          %config InlineBackend.figure_format = 'retina'

          from sklearn import datasets
          import pandas as pd
          import numpy as np
          import matplotlib.pyplot as plt

          plt.style.use('seaborn-white')
```

1.2 Numerical EDA

In this chapter, you'll be working with a dataset obtained from the UCI Machine Learning Repository consisting of votes made by US House of Representatives Congressmen. Your goal will be to predict their party affiliation ('Democrat' or 'Republican') based on how they voted on certain key issues. Here, it's worth noting that we have preprocessed this dataset to deal with missing values. This is so that your focus can be directed towards understanding how to train and evaluate supervised learning models. Once you have mastered these fundamentals, you will be introduced to preprocessing techniques in Chapter 4 and have the chance to apply them there yourself - including on this very same dataset!

Before thinking about what supervised learning models you can apply to this, however, you need to perform Exploratory data analysis (EDA) in order to understand the structure of the data. For a refresher on the importance of EDA, check out the first two chapters of Statistical Thinking in Python (Part 1).

Get started with your EDA now by exploring this voting records dataset numerically. It has been pre-loaded for you into a DataFrame called `df`. Use pandas' `.head()`, `.info()`, and `.describe()` methods in the IPython Shell to explore the DataFrame, and select the statement below that is **not** true.

Possible Answers

- The DataFrame has a total of 435 rows and 17 columns.
- Except for 'party', all of the columns are of type int64.

- The first two rows of the DataFrame consist of votes made by Republicans and the next three rows consist of votes made by Democrats.
- There are 17 predictor variables, or features, in this DataFrame.
- The target variable in this DataFrame is 'party'.

```
In [297]: votes_df = pd.read_csv('data/house-votes-84-clean.csv')
```

```
In [298]: votes_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 435 entries, 0 to 434
Data columns (total 17 columns):
party                435 non-null object
infants              435 non-null int64
water                435 non-null int64
budget               435 non-null int64
physician            435 non-null int64
salvador             435 non-null int64
religious            435 non-null int64
satellite            435 non-null int64
aid                  435 non-null int64
missile              435 non-null int64
immigration          435 non-null int64
synfuels             435 non-null int64
education            435 non-null int64
superfund            435 non-null int64
crime                435 non-null int64
duty_free_exports    435 non-null int64
eaa_rsa              435 non-null int64
dtypes: int64(16), object(1)
memory usage: 57.9+ KB
```

```
In [299]: votes_df.head()
```

```
Out[299]:         party  infants  water  budget  physician  salvador  religious  \
          0  republican        0      1       0          1         1          1
          1  republican        0      1       0          1         1          1
          2    democrat        0      1       1          0         1          1
          3    democrat        0      1       1          0         1          1
          4    democrat        1      1       1          0         1          1

             satellite  aid  missile  immigration  synfuels  education  superfund  \
          0          0    0        0            1         0          1          1
          1          0    0        0            0         0          1          1
          2          0    0        0            0         1          0          1
          3          0    0        0            0         1          0          1
          4          0    0        0            0         1          0          1
```

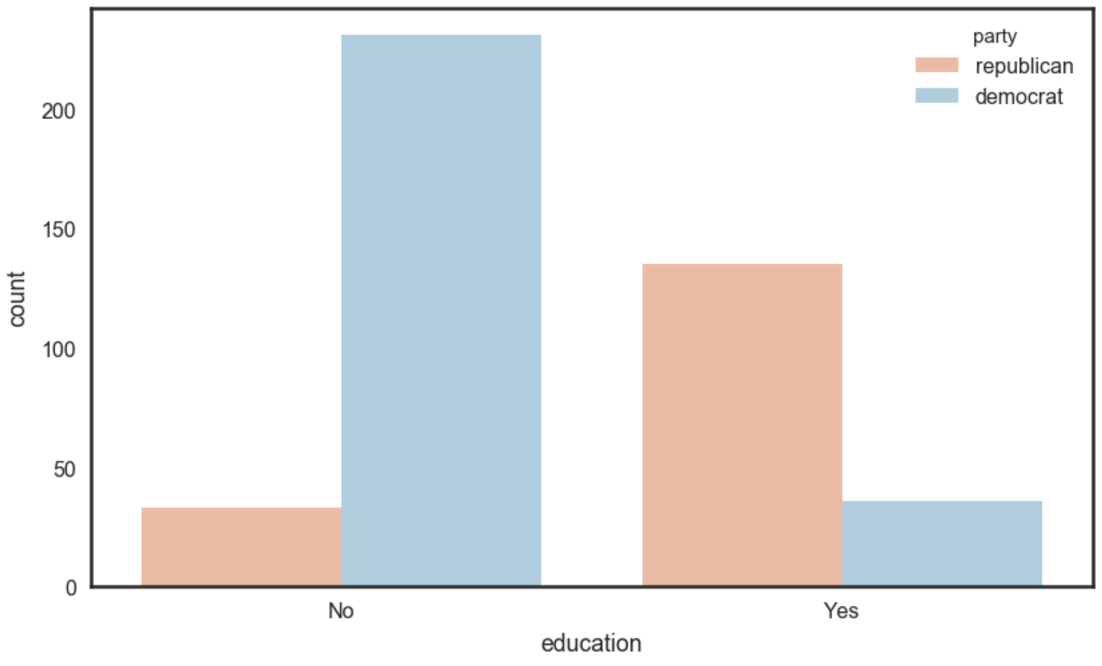
```
   crime  duty_free_exports  eaa_rsa
0      1                  0        1
1      1                  0        1
2      1                  0        0
3      0                  0        1
4      1                  1        1
```

1.3 Visual NDA

The Numerical EDA you did in the previous exercise gave you some very important information, such as the names and data types of the columns, and the dimensions of the DataFrame. Following this with some visual EDA will give you an even better understanding of the data. In the video, Hugo used the `scatter_matrix()` function on the Iris data for this purpose. However, you may have noticed in the previous exercise that all the features in this dataset are binary; that is, they are either 0 or 1. So a different type of plot would be more useful here, such as Seaborn's `countplot`.

Given on the right is a `countplot` of the `'education'` bill, generated from the following code:

```
In [300]: plt.figure()
          sns.countplot(x='education', hue='party', data=votes_df, palette='RdBu')
          plt.xticks([0,1], ['No', 'Yes'])
          plt.show()
```

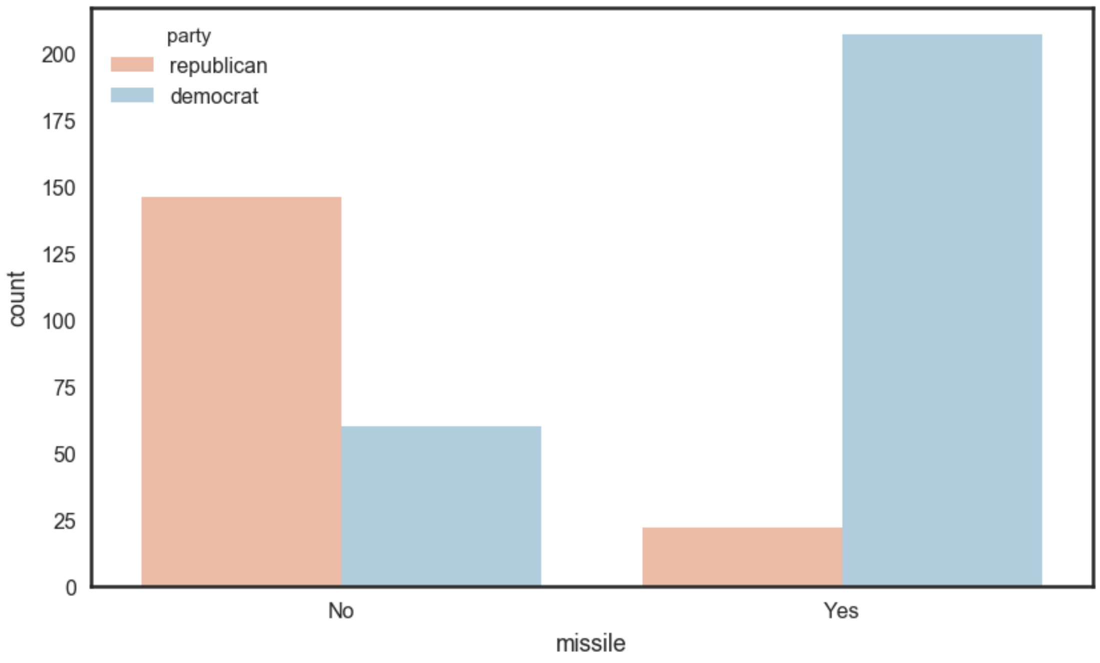


In `sns.countplot()`, we specify the x-axis data to be `'education'`, and hue to be `'party'`. Recall that 'party' is also our target variable. So the resulting plot shows the difference in voting behavior between the two parties for the `'education'` bill, with each party colored differently. We manually specified the color to be `'RdBu'`, as the Republican party has been traditionally associated with red, and the Democratic party with blue.

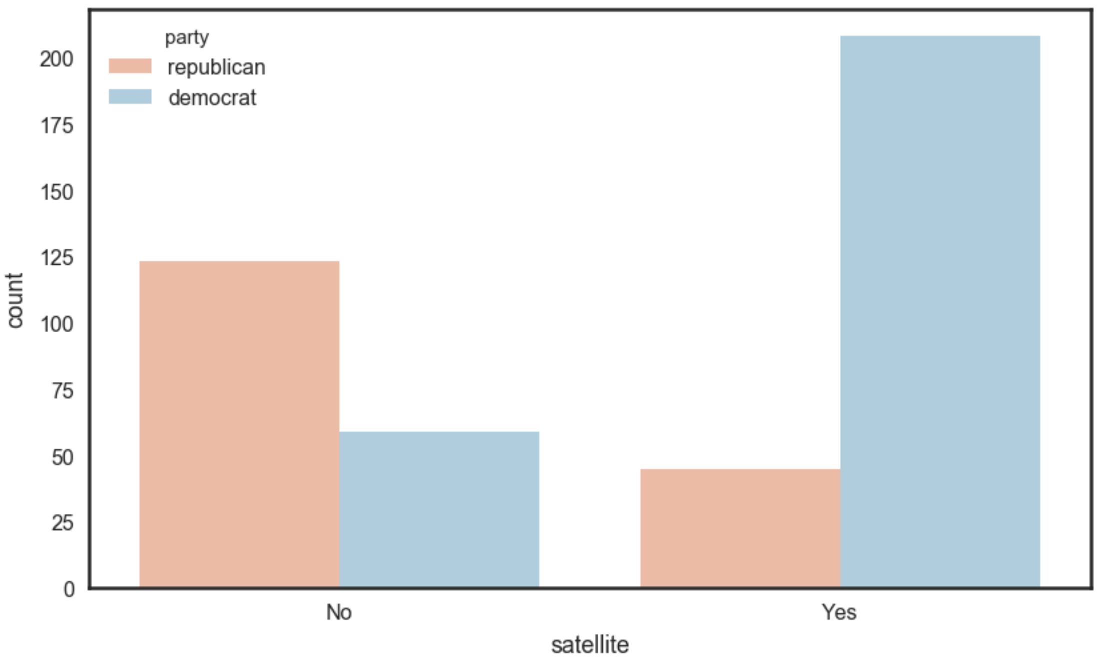
It seems like Democrats voted resoundingly against this bill, compared to Republicans. This is the kind of information that our machine learning model will seek to learn when we try to predict party affiliation solely based on voting behavior. An expert in U.S politics may be able to predict this without machine learning, but probably not instantaneously - and certainly not if we are dealing with hundreds of samples!

In the IPython Shell, explore the voting behavior further by generating countplots for the `'satellite'` and `'missile'` bills, and answer the following question: Of these two bills, for which ones do Democrats vote resoundingly in favor of, compared to Republicans? Be sure to begin your plotting statements for each figure with `plt.figure()` so that a new figure will be set up. Otherwise, your plots will be overlayed onto the same figure.

```
In [301]: plt.figure()
          sns.countplot(x='missile', hue='party', data=votes_df, palette='RdBu')
          plt.xticks([0,1], ['No', 'Yes'])
          plt.show()
```



```
In [302]: plt.figure()
          sns.countplot(x='satellite', hue='party', data=votes_df, palette='RdBu')
          plt.xticks([0,1], ['No', 'Yes'])
          plt.show()
```



1.4 The classification challenge (video)

1.5 k-Nearest Neighbors: Fit

Having explored the Congressional voting records dataset, it is time now to build your first classifier. In this exercise, you will fit a k-Nearest Neighbors classifier to the voting dataset, `votes_df`.

In the video, Hugo discussed the importance of ensuring your data adheres to the format required by the scikit-learn API. The features need to be in an array where each column is a feature and each row a different observation or data point - in this case, a Congressman's voting record. The target needs to be a single column with the same number of observations as the feature data. We have done this for you in this exercise. Notice we named the feature array X and response variable y: This is in accordance with the common scikit-learn practice.

Your job is to create an instance of a k-NN classifier with 6 neighbors (by specifying the n_neighbors parameter) and then fit it to the data.

```
In [303]: # Import KNeighborsClassifier from sklearn.neighbors
          from sklearn.neighbors import KNeighborsClassifier

          # Create arrays for the features and the response variable
          y = votes_df['party'].values
          X = votes_df.drop('party', axis=1).values

          # Create a k-NN classifier with 6 neighbors
          knn = KNeighborsClassifier(n_neighbors = 6)
          # Fit the classifier to the data
          knn.fit(X,y)

Out[303]: KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
                     metric_params=None, n_jobs=1, n_neighbors=6, p=2,
                     weights='uniform')
```

1.6 k-Nearest Neighbors: Predict

Having fit a k-NN classifier, you can now use it to predict the label of a new data point. However, there is no unlabeled data available since all of it was used to fit the model! You can still use the .predict() method on the X that was used to fit the model, but it is not a good indicator of the model's ability to generalize to new, unseen data.

In the next video, Hugo will discuss a solution to this problem. For now, a random unlabeled data point has been generated and is available to you as X_new. You will use your classifier to predict the label for this new data point, as well as on the training data X that the model has already seen. Using .predict() on X_new will generate 1 prediction, while using it on X will generate 435 predictions: 1 for each sample.

The DataFrame has been pre-loaded as df. This time, you will create the feature array X and target variable array y yourself.

```
In [304]: X_new = pd.read_csv('data/voting-features.csv')

          # Predict the labels for the training data X
          y_pred = knn.predict(X)

          # Predict and print the label for the new data point X_new
          new_prediction = knn.predict(X_new)
          print("Prediction: {}".format(new_prediction))

Prediction: ['democrat']
```

1.7 The digits recognition dataset

Up until now, you have been performing binary classification, since the target variable had two possible outcomes. Hugo, however, got to perform multi-class classification in the videos, where the target variable could take on three possible outcomes. Why does he get to have all the fun?! In the following exercises, you'll be working with the MNIST digits recognition dataset, which has 10 classes, the digits 0 through 9! A reduced version of the MNIST dataset is one of scikit-learn's included datasets, and that is the one we will use in this exercise.

Each sample in this scikit-learn dataset is an 8x8 image representing a handwritten digit. Each pixel is represented by an integer in the range 0 to 16, indicating varying levels of black. Recall that scikit-learn's built-in datasets are of type Bunch, which are dictionary-like objects. Helpfully for the MNIST dataset, scikit-learn provides an `'images'` key in addition to the `'data'` and `'target'` keys that you have seen with the Iris data. Because it is a 2D array of the images corresponding to each sample, this `'images'` key is useful for visualizing the images, as you'll see in this exercise (for more on plotting 2D arrays, see Chapter 2 of DataCamp's course on Data Visualization with

Python). On the other hand, the 'data' key contains the feature array - that is, the images as a flattened array of 64 pixels.

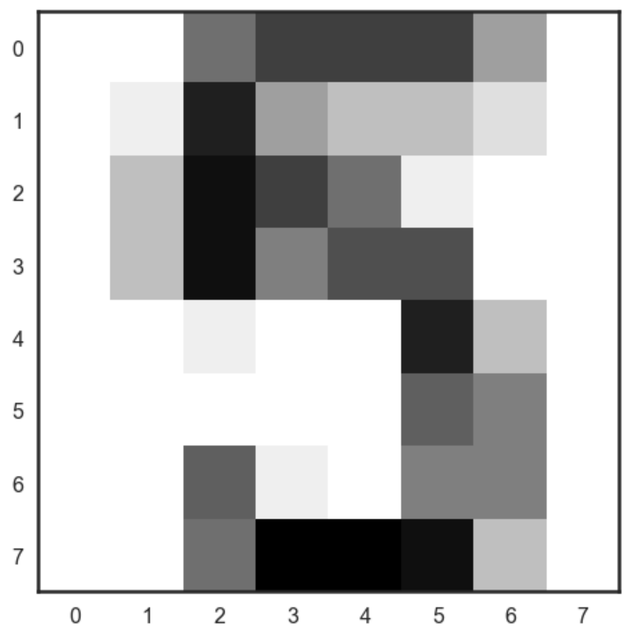
Notice that you can access the keys of these Bunch objects in two different ways: By using the . notation, as in `digits.images`, or the `[]` notation, as in `digits['images']`.

For more on the MNIST data, check out this exercise in Part 1 of DataCamp's Importing Data in Python course. There, the full version of the MNIST dataset is used, in which the images are 28x28. It is a famous dataset in machine learning and computer vision, and frequently used as a benchmark to evaluate the performance of a new model.

```
In [305]: # Import necessary modules
          from sklearn import datasets
          import matplotlib.pyplot as plt

          # Load the digits dataset: digits
          digits = datasets.load_digits()

          # Print the keys and DESCR of the dataset
          print(digits.keys())
          print(digits.DESCR)

          # Print the shape of the images and data keys
          print(digits.images.shape)
          print(digits.data.shape)

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

```
dict_keys(['data', 'target', 'target_names', 'images', 'DESCR'])
Optical Recognition of Handwritten Digits Data Set
===================================================

Notes
-----
Data Set Characteristics:
    :Number of Instances: 5620
    :Number of Attributes: 64
    :Attribute Information: 8x8 image of integer pixels in the range 0..16.
    :Missing Attribute Values: None
    :Creator: E. Alpaydin (alpaydin '@' boun.edu.tr)
    :Date: July; 1998

This is a copy of the test set of the UCI ML hand-written digits datasets
http://archive.ics.uci.edu/ml/datasets/Optical+Recognition+of+Handwritten+Digits

The data set contains images of hand-written digits: 10 classes where
each class refers to a digit.
```

```
Preprocessing programs made available by NIST were used to extract
normalized bitmaps of handwritten digits from a preprinted form. From a
total of 43 people, 30 contributed to the training set and different 13
to the test set. 32x32 bitmaps are divided into nonoverlapping blocks of
4x4 and the number of on pixels are counted in each block. This generates
an input matrix of 8x8 where each element is an integer in the range
0..16. This reduces dimensionality and gives invariance to small
distortions.

For info on NIST preprocessing routines, see M. D. Garris, J. L. Blue, G.
T. Candela, D. L. Dimmick, J. Geist, P. J. Grother, S. A. Janet, and C.
L. Wilson, NIST Form-Based Handprint Recognition System, NISTIR 5469,
1994.

References
----------
  - C. Kaynak (1995) Methods of Combining Multiple Classifiers and Their
    Applications to Handwritten Digit Recognition, MSc Thesis, Institute of
    Graduate Studies in Science and Engineering, Bogazici University.
  - E. Alpaydin, C. Kaynak (1998) Cascading Classifiers, Kybernetika.
  - Ken Tang and Ponnuthurai N. Suganthan and Xi Yao and A. Kai Qin.
    Linear dimensionalityreduction using relevance weighted LDA. School of
    Electrical and Electronic Engineering Nanyang Technological University.
    2005.
  - Claudio Gentile. A New Approximate Maximal Margin Classification
    Algorithm. NIPS. 2000.

(1797, 8, 8)
(1797, 64)
```



It looks like the image in question corresponds to the digit '5'.

1.8 Train/Test Split + Fit/Predict/Accuracy

Now that you have learned about the importance of splitting your data into training and test sets, it's time to practice doing this on the digits dataset! After creating arrays for the features and target variable, you will split them into training and test sets, fit a k-NN classifier to the training data, and then compute its accuracy using the `.score()` method.

```
In [306]: # Import necessary modules
          from sklearn.neighbors import KNeighborsClassifier
          from sklearn.model_selection import train_test_split

          # Create feature and target arrays
          X = digits.data
          y = digits.target

          # Split into training and test set
          X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, random_stat

          # Create a k-NN classifier with 7 neighbors: knn
          knn = KNeighborsClassifier(n_neighbors = 7)

          # Fit the classifier to the training data
          knn.fit(X_train, y_train)

          # Print the accuracy
          print(knn.score(X_test, y_test))
```

```
0.983333333333
```

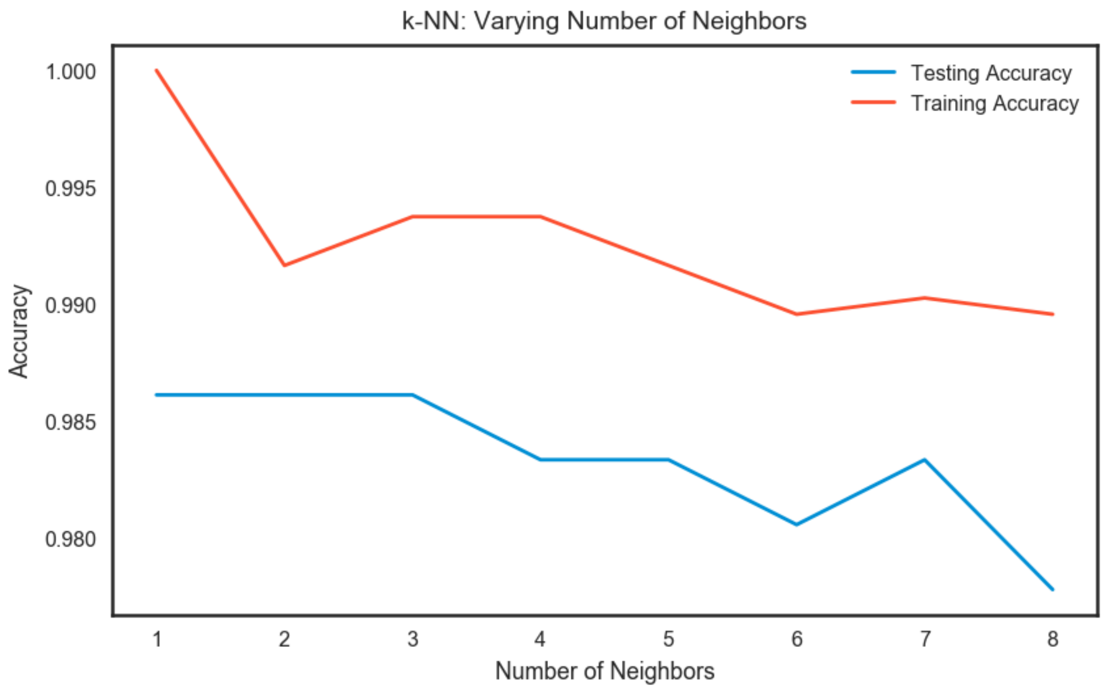
Incredibly, this out of the box k-NN classifier with 7 neighbors has learned from the training data and predicted the labels of the images in the test set with 98% accuracy, and it did so in less than a second! This is one illustration of how incredibly useful machine learning techniques can be.

1.9 Overfitting and underfitting

Remember the model complexity curve that Hugo showed in the video? You will now construct such a curve for the digits dataset! In this exercise, you will compute and plot the training and testing accuracy scores for a variety of different neighbor values. By observing how the accuracy scores differ for the training and testing sets with different values of k, you will develop your intuition for overfitting and underfitting.

The training and testing sets are available to you in the workspace as X_train, X_test, y_train, y_test. In addition, KNeighborsClassifier has been imported from sklearn.neighbors.

```
In [307]: # 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):
              # Setup a k-NN Classifier with k neighbors: knn
              knn = KNeighborsClassifier(n_neighbors = k)

              # Fit the classifier to the training data
              knn.fit(X_train, y_train)

              #Compute accuracy on the training set
              train_accuracy[i] = knn.score(X_train, y_train)

              #Compute accuracy on the testing set
              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.

2 Regression

2.1 Importing data for supervised learning

In this chapter, you will work with Gapminder data that we have consolidated into one CSV file available in the workspace as 'gapminder.csv'. Specifically, your goal will be to use this data to predict the life expectancy in a given country based on features such as the country's GDP, fertility rate, and population. As in Chapter 1, the dataset has been preprocessed.

Since the target variable here is quantitative, this is a regression problem. To begin, you will fit a linear regression with just one feature: 'fertility', which is the average number of children a woman in a given country gives birth to. In later exercises, you will use all the features to build regression models.

Before that, however, you need to import the data and get it into the form needed by scikit-learn. This involves creating feature and target variable arrays. Furthermore, since you are going to use only one feature to begin with, you need to do some reshaping using NumPy's .reshape() method. Don't worry too much about this reshaping right now, but it is something you will have to do occasionally when working with scikit-learn so it is useful to practice.

```
In [309]: # Import numpy and pandas
          import numpy as np
          import pandas as pd
```

```
# Read the CSV file into a DataFrame: df
gm_df = pd.read_csv('data/gapminder.csv')

# Create arrays for features and target variable
y = gm_df['life']
X = gm_df['fertility']

# Print the dimensions of X and y before reshaping
print("Dimensions of y before reshaping: {}".format(y.shape))
print("Dimensions of X before reshaping: {}".format(X.shape))

# Reshape X and y
y = y.reshape(-1,1)
X = X.reshape(-1,1)

# Print the dimensions of X and y after reshaping
print("Dimensions of y after reshaping: {}".format(y.shape))
print("Dimensions of X after reshaping: {}".format(X.shape))
```

```
Dimensions of y before reshaping: (139,)
Dimensions of X before reshaping: (139,)
Dimensions of y after reshaping: (139, 1)
Dimensions of X after reshaping: (139, 1)


/Users/VAN/anaconda/lib/python3.6/site-packages/ipykernel_launcher.py:17: FutureWarning: reshape
/Users/VAN/anaconda/lib/python3.6/site-packages/ipykernel_launcher.py:18: FutureWarning: reshape
```

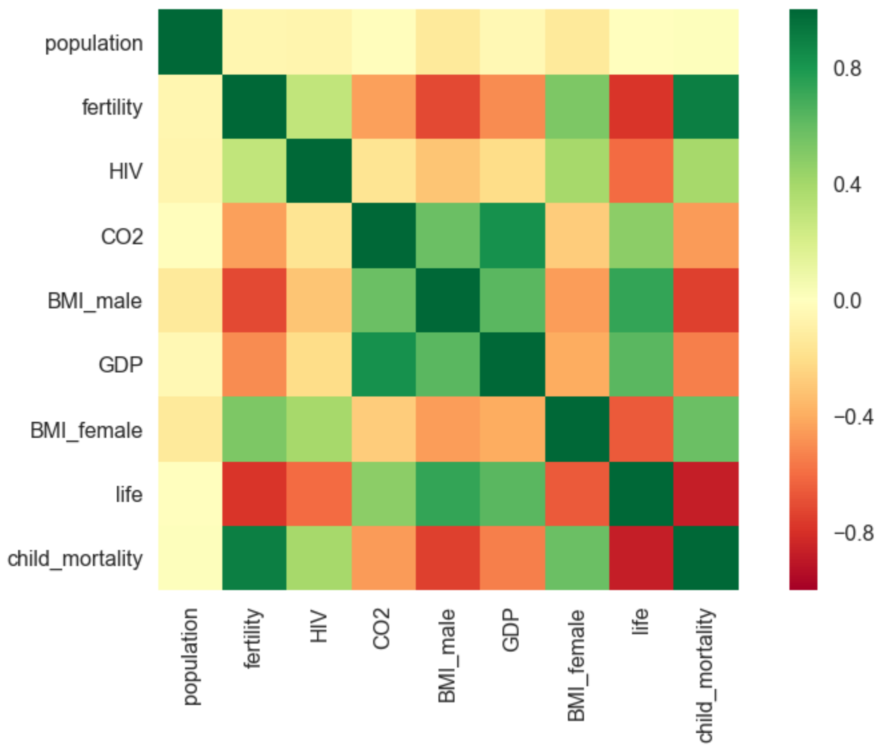
2.2 Exploring the Gapminder data

As always, it is important to explore your data before building models. On the right, we have constructed a heatmap showing the correlation between the different features of the Gapminder dataset, which has been pre-loaded into a DataFrame as df and is available for exploration in the IPython Shell. Cells that are in green show positive correlation, while cells that are in red show negative correlation. Take a moment to explore this: Which features are positively correlated with life, and which ones are negatively correlated? Does this match your intuition?

Then, in the IPython Shell, explore the DataFrame using pandas methods such as .info(), .describe(), .head().

In case you are curious, the heatmap was generated using Seaborn's heatmap function and the following line of code, where df.corr() computes the pairwise correlation between columns:

```
In [311]: sns.heatmap(gm_df.corr(), square=True, cmap='RdYlGn')

Out[311]: <matplotlib.axes._subplots.AxesSubplot at 0x113fa8128>
```



Once you have a feel for the data, consider the statements below and select the one that is not true. After this, Hugo will explain the mechanics of linear regression in the next video and you will be on your way building regression models!

Possible Answers

- The DataFrame has 139 samples (or rows) and 9 columns.
- `life` and `fertility` are negatively correlated.
- The mean of life is 69.602878.
- `fertility` is of type `int64`.
- `GDP` and `life` are positively correlated.

```
In [312]: gm_df.info()

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 139 entries, 0 to 138
Data columns (total 9 columns):
population         139 non-null int64
fertility          139 non-null float64
HIV                139 non-null float64
CO2                139 non-null float64
BMI_male           139 non-null float64
GDP                139 non-null int64
BMI_female         139 non-null float64
```

```
life                 139 non-null float64
child_mortality      139 non-null float64
dtypes: float64(7), int64(2)
memory usage: 9.9 KB
```

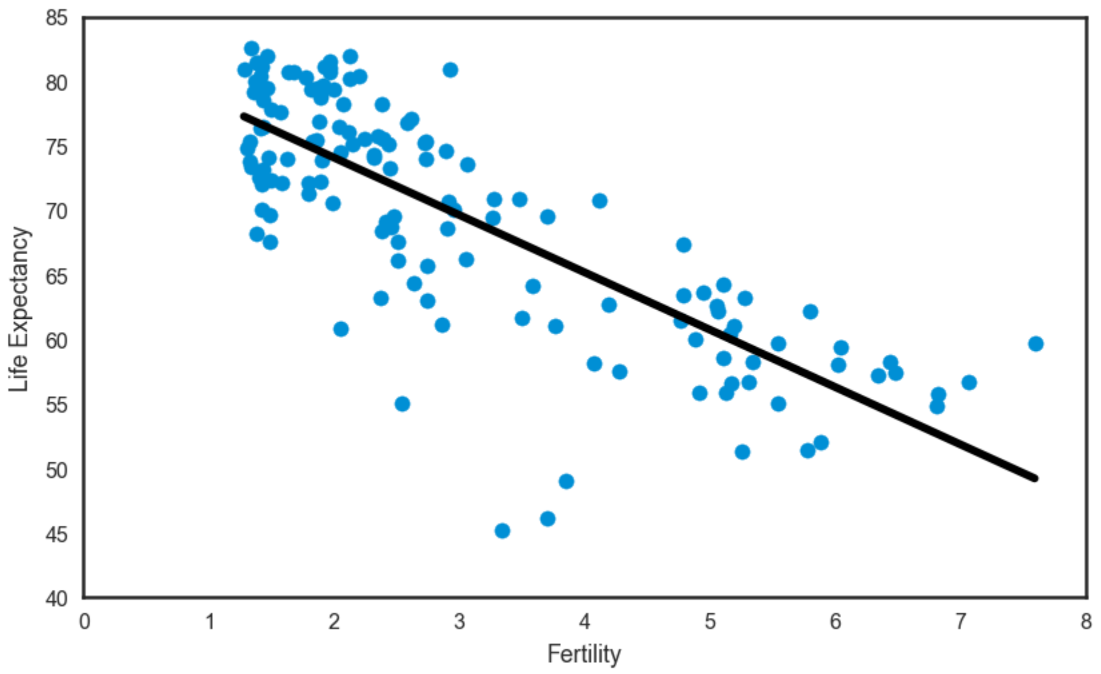
2.3 Fit & predict for regression

Now, you will fit a linear regression and predict life expectancy using just one feature. You saw Andy do this earlier using the `'RM'` feature of the Boston housing dataset. In this exercise, you will use the `'fertility'` feature of the Gapminder dataset. Since the goal is to predict life expectancy, the target variable here is `'life'`.

A scatter plot with `'fertility'` on the x-axis and `'life'` on the y-axis has been generated. As you can see, there is a strongly negative correlation, so a linear regression should be able to capture this trend. Your job is to fit a linear regression and then predict the life expectancy, overlaying these predicted values on the plot to generate a regression line. You will also compute and print the R^2 score using sckit-learn's `.score()` method.

```
In [313]: plt.scatter(X, y)
          plt.xlim(0, 8)
          plt.ylim(40, 85)
          plt.xlabel('Fertility')
          plt.ylabel('Life Expectancy')

          # Import LinearRegression
          from sklearn.linear_model import LinearRegression

          # Create the regressor: reg
          reg = LinearRegression()

          # Create the prediction space
          prediction_space = np.linspace(min(X), max(X)).reshape(-1,1)

          # Fit the model to the data
          reg.fit(X, y)

          # Compute predictions over the prediction space: y_pred
          y_pred = reg.predict(prediction_space)

          # Print R^2
          print(reg.score(X, y))

          # Plot regression line

          plt.plot(prediction_space, y_pred, color='black', linewidth=3)
          plt.show()
```

```
0.619244216774
```



Notice how the line captures the underlying trend in the data. And the performance is quite decent for this basic regression model with only one feature!

2.4 Train/test split for regression

As you learned in Chapter 1, train and test sets are vital to ensure that your supervised learning model is able to generalize well to new data. This was true for classification models, and is equally true for linear regression models.

In this exercise, you will split the Gapminder dataset into training and testing sets, and then fit and predict a linear regression over **all** features. In addition to computing the R^2 score, you will also compute the Root Mean Squared Error (RMSE), which is another commonly used metric to evaluate regression models. The feature array `X` and target variable array `y` have been pre-loaded for you from the DataFrame `df`.

```
In [314]: # Import necessary modules
          from sklearn.linear_model import LinearRegression
          from sklearn.metrics import mean_squared_error
          from sklearn.model_selection import train_test_split

          # Create training and test sets
          X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.3, random_stat

          # Create the regressor: reg_all
          reg_all = LinearRegression()

          # Fit the regressor to the training data
```

```
reg_all.fit(X_train, y_train)

# Predict on the test data: y_pred
y_pred = reg_all.predict(X_test)

# Compute and print R^2 and RMSE
print("R^2: {}".format(reg_all.score(X_test, y_test)))
rmse = np.sqrt(mean_squared_error(y_test, y_pred))
print("Root Mean Squared Error: {}".format(rmse))
```

```
R^2: 0.7298987360907498
Root Mean Squared Error: 4.194027914110239
```

Using all features has improved the model score. This makes sense, as the model has more information to learn from. However, there is one potential pitfall to this process. Can you spot it? You'll learn about this as well how to better validate your models in the next video!

2.5 Cross-validation (video)

2.6 5-fold cross-validation

Cross-validation is a vital step in evaluating a model. It maximizes the amount of data that is used to train the model, as during the course of training, the model is not only trained, but also tested on all of the available data.

In this exercise, you will practice 5-fold cross validation on the Gapminder data. By default, scikit-learn's cross_val_score() function uses R^2 as the metric of choice for regression. Since you are performing 5-fold cross-validation, the function will return 5 scores. Your job is to compute these 5 scores and then take their average.

The DataFrame has been loaded as df and split into the feature/target variable arrays X and y. The modules pandas and numpy have been imported as pd and np, respectively.

```
In [315]: y = gm_df['life'].values
          X = gm_df.drop('life', axis = 1).values

          # Import the necessary modules
          from sklearn.linear_model import LinearRegression
          from sklearn.model_selection import cross_val_score

          # Create a linear regression object: reg
          reg = LinearRegression()

          # Compute 5-fold cross-validation scores: cv_scores
          cv_scores = cross_val_score(reg, X, y, cv = 5)

          # Print the 5-fold cross-validation scores
          print(cv_scores)

          print("Average 5-Fold CV Score: {}".format(cv_scores.mean()))
```

```
[ 0.81720569  0.82917058  0.90214134  0.80633989  0.94495637]
Average 5-Fold CV Score: 0.8599627722769974
```

2.7 K-Fold CV comparison

Cross validation is essential but do not forget that the more folds you use, the more computationally expensive cross-validation becomes. In this exercise, you will explore this for yourself. Your job is to perform 3-fold cross-validation and then 10-fold cross-validation on the Gapminder dataset.

In the IPython Shell, you can use `%timeit` to see how long each 3-fold CV takes compared to 10-fold CV by executing the following cv=3 and cv=10:

`%timeit cross_val_score(reg, X, y, cv = ____)`

pandas and numpy are available in the workspace as pd and np. The DataFrame has been loaded as df and the feature/target variable arrays X and y have been created.

```
In [316]: # Import necessary modules
          from sklearn.linear_model import LinearRegression
          from sklearn.model_selection import cross_val_score

          # Create a linear regression object: reg
          reg = LinearRegression()

          # Perform 3-fold CV
          cvscores_3 = cross_val_score(reg, X, y, cv = 3)
          print(np.mean(cvscores_3))

          # Perform 10-fold CV
          cvscores_10 = cross_val_score(reg, X, y, cv = 10)
          print(np.mean(cvscores_10))
```

```
0.871871278257
0.84361286201
```

2.8 Regularized regression (video)

2.9 Regularization I: Lasso

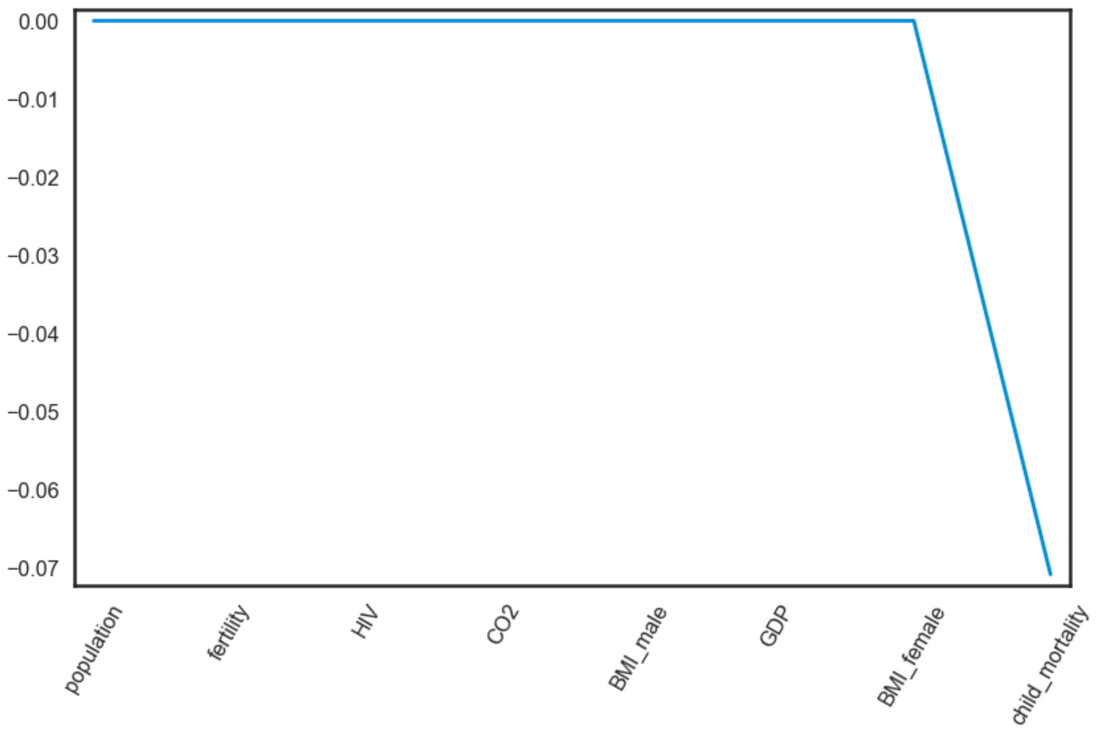
In the video, you saw how Lasso selected out the `'RM'` feature as being the most important for predicting Boston house prices, while shrinking the coefficients of certain other features to 0. Its ability to perform feature selection in this way becomes even more useful when you are dealing with data involving thousands of features.

In this exercise, you will fit a lasso regression to the Gapminder data you have been working with and plot the coefficients. Just as with the Boston data, you will find that the coefficients of some features are shrunk to 0, with only the most important ones remaining.

The feature and target variable arrays have been pre-loaded as `X` and `y`.

```
In [317]: # Import Lasso
          from sklearn.linear_model import Lasso

          # Instantiate a lasso regressor: lasso
          lasso = Lasso(alpha = 0.4, normalize = True)

          # Fit the regressor to the data
          lasso.fit(X, y)

          # Compute and print the coefficients
          lasso_coef = lasso.coef_
          print(lasso_coef)

          # Plot the coefficients
          df_columns = df.drop('life', axis = 1).columns
          plt.plot(range(len(df_columns)), lasso_coef)
          plt.xticks(range(len(df_columns)), df_columns.values, rotation=60)
          plt.margins(0.02)
          plt.show()
```

```
[-0.         -0.         -0.          0.          0.          0.         -0.
 -0.07087587]
```



According to the lasso algorithm, it seems like `'child_mortality'` is the most important feature when predicting life expectancy.

2.10 Regularization II: Ridge

Lasso is great for feature selection, but when building regression models, Ridge regression should be your first choice.

Recall that lasso performs regularization by adding to the loss function a penalty term of the absolute value of each coefficient multiplied by some alpha. This is also known as $L1$ regularization because the regularization term is the $L1$ norm of the coefficients. This is not the only way to regularize, however.

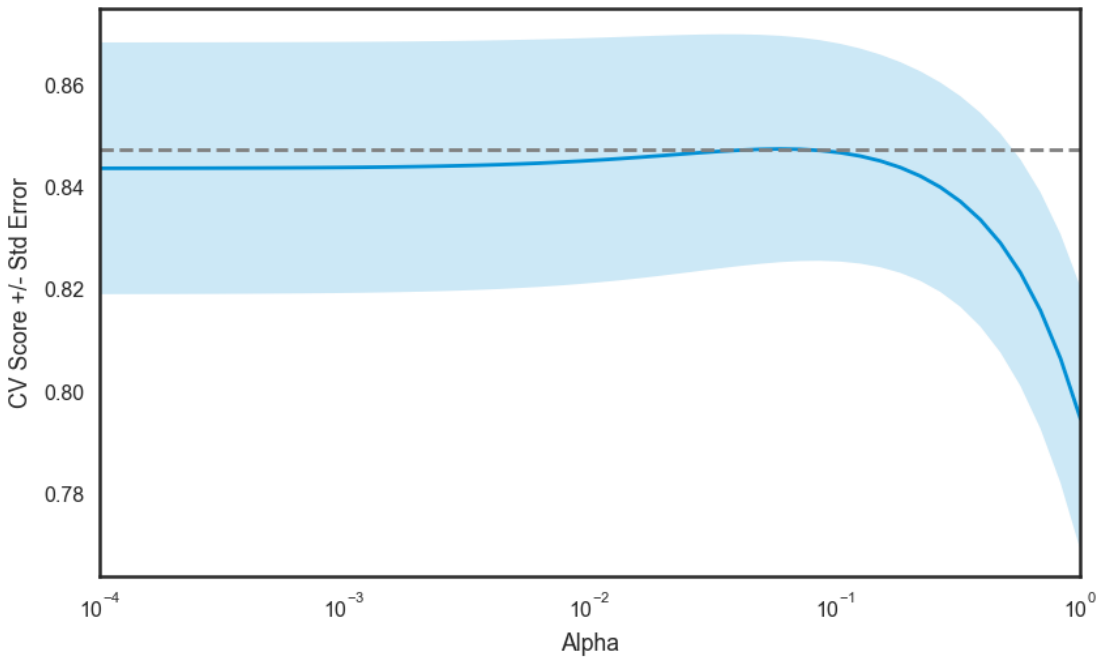
If instead you took the sum of the squared values of the coefficients multiplied by some alpha - like in Ridge regression - you would be computing the $L2$ norm. In this exercise, you will practice fitting ridge regression models over a range of different alphas, and plot cross-validated R^2 scores for each, using this function that we have defined for you, which plots the R^2 score as well as standard error for each alpha:

```
In [240]: def display_plot(cv_scores, cv_scores_std):
              fig = plt.figure()
              ax = fig.add_subplot(1,1,1)
              ax.plot(alpha_space, cv_scores)

              std_error = cv_scores_std / np.sqrt(10)

              ax.fill_between(alpha_space, cv_scores + std_error, cv_scores - std_error, alpha=0
              ax.set_ylabel('CV Score +/- Std Error')
              ax.set_xlabel('Alpha')
              ax.axhline(np.max(cv_scores), linestyle='--', color='.5')
              ax.set_xlim([alpha_space[0], alpha_space[-1]])
              ax.set_xscale('log')
              plt.show()
```

Don't worry about the specifics of the above function works. The motivation behind this exercise is for you to see how the R^2 score varies with different alphas, and to understand the importance of selecting the right value for alpha. You'll learn how to tune alpha in the next chapter.

```
In [318]: # Import necessary modules
          from sklearn.linear_model import Ridge
          from sklearn.model_selection import cross_val_score

          # Setup the array of alphas and lists to store scores
          alpha_space = np.logspace(-4, 0, 50)
          ridge_scores = []
          ridge_scores_std = []

          # Create a ridge regressor: ridge
          ridge = Ridge(normalize = True)

          # Compute scores over range of alphas
```

```
for alpha in alpha_space:

    # Specify the alpha value to use: ridge.alpha
    ridge.alpha = alpha

    # Perform 10-fold CV: ridge_cv_scores
    ridge_cv_scores = cross_val_score(ridge, X, y, cv = 10)

    # Append the mean of ridge_cv_scores to ridge_scores
    ridge_scores.append(np.mean(ridge_cv_scores))

    # Append the std of ridge_cv_scores to ridge_scores_std
    ridge_scores_std.append(np.std(ridge_cv_scores))

# Display the plot
display_plot(ridge_scores, ridge_scores_std)
```



Notice how the cross-validation scores change with different alphas. Which alpha should you pick? How can you fine-tune your model? You'll learn all about this in the next chapter!

3 Fine-tuning your model

3.1 Metrics for classification

In Chapter 1, you evaluated the performance of your k-NN classifier based on its accuracy. However, as Andy discussed, accuracy is not always an informative metric. In this exercise, you will dive more deeply into evaluating the performance of binary classifiers by computing a confusion matrix and generating a classification report.

You may have noticed in the video that the classification report consisted of three rows, and an additional support column. The support gives the number of samples of the true response that lie in that class - so in the video example, the support was the number of Republicans or Democrats in the test set on which the classification report was computed. The precision, recall, and f1-score columns, then, gave the respective metrics for that particular class.

Here, you'll work with the PIMA Indians dataset obtained from the UCI Machine Learning Repository. The goal is to predict whether or not a given female patient will contract diabetes based on features such as BMI, age, and number of pregnancies. Therefore, it is a binary classification problem. A target value of 0 indicates that the patient does not have diabetes, while a value of 1 indicates that the patient does have diabetes. As in Chapters 1 and 2, the dataset has been preprocessed to deal with missing values.

The dataset has been loaded into a DataFrame `df` and the feature and target variable arrays `X` and `y` have been created for you. In addition, `sklearn.model_selection.train_test_split` and `sklearn.neighbors.KNeighborsClassifier` have already been imported.

Your job is to train a k-NN classifier to the data and evaluate its performance by generating a confusion matrix and classification report.

```
In [320]: diabetes_df = pd.read_csv('data/diabetes-clean.csv')
```

```
In [321]: X = diabetes_df.drop('diabetes', axis = 1).values
          y = diabetes_df['diabetes']

          # Import necessary modules
          from sklearn.metrics import classification_report, confusion_matrix

          # Create training and test set
          X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.4, random_stat

          # Instantiate a k-NN classifier: knn
          knn = KNeighborsClassifier(n_neighbors = 6)

          # Fit the classifier to the training data
          knn.fit(X_train, y_train)

          # Predict the labels of the test data: y_pred
          y_pred = knn.predict(X_test)

          # Generate the confusion matrix and classification report
          print(confusion_matrix(y_test, y_pred))
          print(classification_report(y_test, y_pred))
```

```
[[176  30]
 [ 52  50]]
             precision    recall  f1-score   support

          0       0.77      0.85      0.81       206
          1       0.62      0.49      0.55       102

avg / total       0.72      0.73      0.72       308
```

3.2 Logistic regression and the ROC curve (video)

3.3 Building a logistic regression model

Time to build your first logistic regression model! As Hugo showed in the video, scikit-learn makes it very easy to try different models, since the Train-Test-Split/Instantiate/Fit/Predict paradigm applies to all classifiers and regressors - which are known in scikit-learn as 'estimators'. You'll see this now for yourself as you train a logistic regression model on exactly the same data as in the previous exercise. Will it outperform k-NN? There's only one way to find out!

The feature and target variable arrays `X` and `y` have been pre-loaded, and `train_test_split` has been imported for you from `sklearn.model_selection`.

```
In [322]: # Import the necessary modules
          from sklearn.linear_model import LogisticRegression
          from sklearn.metrics import confusion_matrix, classification_report

          # Create training and test sets
          X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.4, random_stat

          # Create the classifier: logreg
          logreg = LogisticRegression()

          # Fit the classifier to the training data
          logreg.fit(X_train, y_train)

          # Predict the labels of the test set: y_pred
          y_pred = logreg.predict(X_test)

          # Compute and print the confusion matrix and classification report
          print(confusion_matrix(y_test, y_pred))
          print(classification_report(y_test, y_pred))
```

```
[[176  30]
 [ 35  67]]
             precision    recall  f1-score   support

          0       0.83      0.85      0.84       206
          1       0.69      0.66      0.67       102

avg / total       0.79      0.79      0.79       308
```

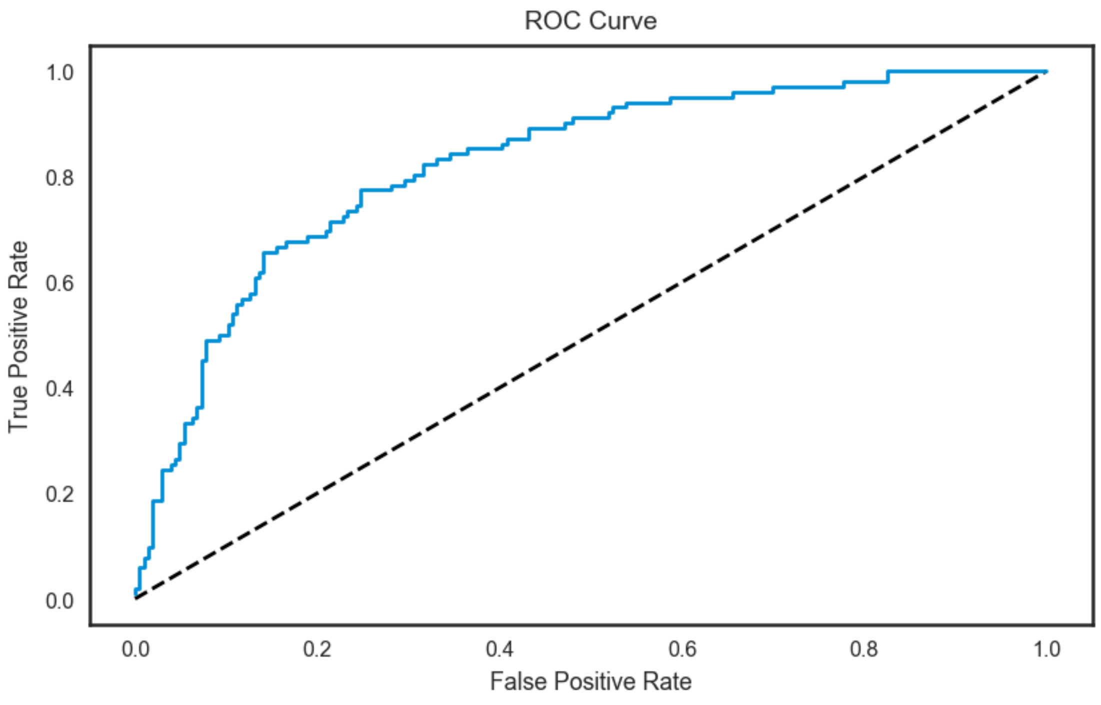
3.4 Plotting an ROC curve

Great job in the previous exercise - you now have a new addition to your toolbox of classifiers!

Classification reports and confusion matrices are great methods to quantitatively evaluate model performance, while ROC curves provide a way to visually evaluate models. As Hugo demonstrated in the video, most classifiers in scikit-learn have a `.predict_proba()` method which returns the probability of a given sample being in a particular class. Having built a logistic regression model, you'll now evaluate its performance by plotting an ROC curve. In doing so, you'll make use of the `.predict_proba()` method and become familiar with its functionality.

Here, you'll continue working with the PIMA Indians diabetes dataset. The classifier has already been fit to the training data and is available as `logreg`.

```
In [323]: # Import necessary modules
          from sklearn.metrics import roc_curve

          # Compute predicted probabilities: y_pred_prob
          y_pred_prob = logreg.predict_proba(X_test)[:,1]

          # Generate ROC curve values: fpr, tpr, thresholds
          fpr, tpr, thresholds = roc_curve(y_test, y_pred_prob)

          # Plot ROC curve
          plt.plot([0, 1], [0, 1], 'k--')
          plt.plot(fpr, tpr)
          plt.xlabel('False Positive Rate')
          plt.ylabel('True Positive Rate')
          plt.title('ROC Curve')
          plt.show()
```



3.5 Precision-recall Curve

When looking at your ROC curve, you may have noticed that the y-axis (True positive rate) is also known as recall. Indeed, in addition to the ROC curve, there are other ways to visually evaluate model performance. One such way is the precision-recall curve, which is generated by plotting the precision and recall for different thresholds. As a reminder, precision and recall are defined as:

$$Precision = \frac{TP}{TP + FP}$$

$$Recall = \frac{TP}{TP + FN}$$

On the right, a precision-recall curve has been generated for the diabetes dataset. The classification report and confusion matrix are displayed in the IPython Shell.

Study the precision-recall curve and then consider the statements given below. Choose the one statement that is not true. Note that here, the class is positive (1) if the individual has diabetes.

Possible Answers

- A recall of 1 corresponds to a classifier with a low threshold in which all females who contract diabetes were correctly classified as such, at the expense of many misclassifications of those who did not have diabetes. (This is actually a true statement! Observe how when the recall is high, the precision drops.)
- Precision is undefined for a classifier which makes no positive predictions, that is, classifies everyone as not having diabetes. (In the case when there are no true positives or true negatives, precision is $0/0$, which is undefined.)
- When the threshold is very close to 1, precision is also 1, because the classifier is absolutely certain about its predictions. (This is a correct statement. Notice how a high precision corresponds to a low recall: The classifier has a high threshold to ensure the positive predictions it makes are correct, which means it may miss some positive labels that have lower probabilities.)
- Precision and recall take true negatives into consideration.

3.6 Area under the ROC curve (video)

3.7 AUC computation

Say you have a binary classifier that in fact is just randomly making guesses. It would be correct approximately 50% of the time, and the resulting ROC curve would be a diagonal line in which the True Positive Rate and False Positive Rate are always equal. The Area under this ROC curve would be 0.5. This is one way in which the AUC, which Hugo discussed in the video, is an informative metric to evaluate a model. If the AUC is greater than 0.5, the model is better than random guessing. Always a good sign!

In this exercise, you'll calculate AUC scores using the roc_auc_score() function from sklearn.metrics as well as by performing cross-validation on the diabetes dataset.

Training and test sets X_train, X_test, y_train, y_test have been pre-loaded for you, and a logistic regression classifier logreg has been fit to the training data.

```
In [324]: # Import necessary modules
          from sklearn.metrics import roc_auc_score
          from sklearn.model_selection import cross_val_score

          # Compute predicted probabilities: y_pred_prob
          y_pred_prob = logreg.predict_proba(X_test)[:,1]

          # Compute and print AUC score
          print("AUC: {}".format(roc_auc_score(y_test, y_pred_prob)))

          # Compute cross-validated AUC scores: cv_auc
          cv_auc = cross_val_score(logreg, X, y, cv = 5, scoring = 'roc_auc')

          # Print list of AUC scores
          print("AUC scores computed using 5-fold cross-validation: {}".format(cv_auc))
```

```
AUC: 0.8253379021511518
AUC scores computed using 5-fold cross-validation: [ 0.80185185  0.8062963   0.81481481  0.86245
```

3.8 Hyperparameter tuning (video)

3.9 Hyperparameter tuning with GridSearchCV

Hugo demonstrated how to use to tune the `n_neighbors` parameter of the `KNeighborsClassifier()` using GridSearchCV on the voting dataset. You will now practice this yourself, but by using logistic regression on the diabetes dataset instead!

Like the alpha parameter of lasso and ridge regularization that you saw earlier, logistic regression also has a regularization parameter: C. C controls the inverse of the regularization strength, and this is what you will tune in this exercise. A large C can lead to an overfit model, while a small C can lead to an underfit model.

The hyperparameter space for C has been setup for you. Your job is to use GridSearchCV and logistic regression to find the optimal C in this hyperparameter space. The feature array is available as `X` and target variable array is available as `y`.

You may be wondering why you aren't asked to split the data into training and test sets. Good observation! Here, we want you to focus on the process of setting up the hyperparameter grid and performing grid-search cross-validation. In practice, you will indeed want to hold out a portion of your data for evaluation purposes, and you will learn all about this in the next video!

```
In [325]: # Import necessary modules
          from sklearn.linear_model import LogisticRegression
          from sklearn.model_selection import GridSearchCV

          # Setup the hyperparameter grid
          c_space = np.logspace(-5, 8, 15)
          param_grid = {'C': c_space}

          # Instantiate a logistic regression classifier: logreg
          logreg = LogisticRegression()
```

```
# Instantiate the GridSearchCV object: logreg_cv
logreg_cv = GridSearchCV(logreg, param_grid, cv=5)

# Fit it to the data
logreg_cv.fit(X, y)

# Print the tuned parameters and score
print("Tuned Logistic Regression Parameters: {}".format(logreg_cv.best_params_))
print("Best score is {}".format(logreg_cv.best_score_))
```

```
Tuned Logistic Regression Parameters: {'C': 3.7275937203149381}
Best score is 0.7708333333333334
```

3.10 Hyperparameter tuning with RandomizedSearchCV

GridSearchCV can be computationally expensive, especially if you are searching over a large hyperparameter space and dealing with multiple hyperparameters. A solution to this is to use RandomizedSearchCV, in which not all hyperparameter values are tried out. Instead, a fixed number of hyperparameter settings is sampled from specified probability distributions. You'll practice using RandomizedSearchCV in this exercise and see how this works.

Here, you'll also be introduced to a new model: the Decision Tree. Don't worry about the specifics of how this model works. Just like k-NN, linear regression, and logistic regression, decision trees in scikit-learn have .fit() and .predict() methods that you can use in exactly the same way as before. Decision trees have many parameters that can be tuned, such as max_features, max_depth, and min_samples_leaf: This makes it an ideal use case for RandomizedSearchCV.

As before, the feature array X and target variable array y of the diabetes dataset have been pre-loaded. The hyperparameter settings have been specified for you. Your goal is to use RandomizedSearchCV to find the optimal hyperparameters. Go for it!

```
In [326]: # Import necessary modules
          from scipy.stats import randint
          from sklearn.tree import DecisionTreeClassifier
          from sklearn.model_selection import RandomizedSearchCV

          # Setup the parameters and distributions to sample from: param_dist
          param_dist = {"max_depth": [3, None],
                        "max_features": randint(1, 9),
                        "min_samples_leaf": randint(1, 9),
                        "criterion": ["gini", "entropy"]}

          # Instantiate a Decision Tree classifier: tree
          tree = DecisionTreeClassifier()

          # Instantiate the RandomizedSearchCV object: tree_cv
          tree_cv = RandomizedSearchCV(tree, param_dist, cv=5)
```

```
# Fit it to the data
tree_cv.fit(X,y)

# Print the tuned parameters and score
print("Tuned Decision Tree Parameters: {}".format(tree_cv.best_params_))
print("Best score is {}".format(tree_cv.best_score_))
```

```
Tuned Decision Tree Parameters: {'criterion': 'entropy', 'max_depth': 3, 'max_features': 6, 'min
Best score is 0.7395833333333334
```

Note that RandomizedSearchCV will never outperform GridSearchCV. Instead, it is valuable because it saves on computation time.

```
In [ ]:
```