SUDHEER TALLURI

Karlijn Willems
February 25th, 2019

MUST READ PYTHON +1

Python Machine Learning: Scikit-Learn Tutorial

An easy-to-follow scikit-learn tutorial that will help you get started with Python machine learning.

Machine Learning with Python

Machine learning is a branch in computer science that studies the design of algorithms that can learn.

Typical tasks are concept learning, function learning or "predictive modeling", clustering and finding predictive patterns. These tasks are learned through available data that were observed through experiences or instructions, for example.

The hope that comes with this discipline is that including the experience into its tasks will eventually improve the learning. But this improvement needs to happen in such a way that the learning itself becomes automatic so that humans like ourselves don't need to interfere anymore is the ultimate goal.

Today's scikit-learn tutorial will introduce you to the basics of Python machine learning:

- You'll learn how to use Python and its libraries to explore your data with the help of `matplotlib` and Principal Component Analysis (PCA),
- And you'll preprocess your data with normalization, and you'll split your data into training and test sets.

built.

- As an extra, you'll also see how you can also use Support Vector Machines (SVM) to construct another model to classify your data.

If you're more interested in an R tutorial, take a look at our Machine Learning with R for Beginners tutorial.

Alternatively, check out DataCamp's Supervised Learning with scikit-learn and Unsupervised Learning in Python courses!

Loading Your Data Set

The first step to about anything in data science is loading your data. This is also the starting point of this scikit-learn tutorial.

This discipline typically works with observed data. This data might be collected by yourself, or you can browse through other sources to find data sets. But if you're not a researcher or otherwise involved in experiments, you'll probably do the latter.

If you're new to this and you want to start problems on your own, finding these data sets might prove to be a challenge. However, you can typically find good data sets at the UCI Machine Learning Repository or on the Kaggle website. Also, check out this KD Nuggets list with resources.

For now, you should warm up, not worry about finding any data by yourself and just load in the `digits` data set that comes with a Python library, called `scikit-learn`.

Fun fact: did you know the name originates from the fact that this library is a scientific toolbox built around SciPy? By the way, there is more than just one scikit out there. This scikit contains modules specifically for machine learning and data mining, which explains the second component of the library name. :)

To load in the data, you import the module `datasets` from `sklearn`. Then, you can use the `load_digits()` method from `datasets` to load in the data:

```
from sklearn import ________

# Load in the `digits` data
digits = datasets.load_digits()

# Print the `digits` data
print(______)
```

Solution Run

Note that the `datasets` module contains other methods to load and fetch popular reference datasets, and you can also count on this module in case you need artificial data generators. Also, this data set is also available through the UCI Repository that was mentioned above: you can find the data here.

If you had decided to pull the data from the latter page, your data import would've looked like this:

script.py IPython Shell

```
# Import the `pandas` library as `pd`
import ______ as __

# Load in the data with `read_csv()`
digits = pd.read_csv("http://archive.ics.uci.edu/ml/machine-learning-databases
/optdigits/optdigits.tra", header=None)

# Print out `digits`
print(______)
```

Solution Run

Note that if you download the data like this, the data is already split up in a training and a test set, indicated by the extensions `.tra` and `.tes`. You'll need to load in both files to elaborate your project. With the command above, you only load in the training set.

Explore Your Data

When first starting out with a data set, it's always a good idea to go through the data description and see what you can already learn. When it comes to `scikit-learn`, you don't immediately have this information readily available, but in the case where you import data from another source, there's usually a data description present, which will already be a sufficient amount of information to gather some insights into your data.

However, these insights are not merely deep enough for the analysis that you are going to perform. You really need to have a good working knowledge about the data set.

Performing an exploratory data analysis (EDA) on a data set like the one that this tutorial now has might seem difficult.

Where do you start exploring these handwritten digits?

Gathering Basic Information on Your Data

Let's say that you haven't checked any data description folder (or maybe you want to double-check the information that has been given to you).

Then you should start by gathering the necessary information.

When you printed out the `digits` data after having loaded it with the help of the `scikit-learn` `datasets` module, you will have noticed that there is already a lot of

Similarly, you can also access the target values or labels through the `target` attribute and the description through the `DESCR` attribute.

To see which keys you have available to already get to know your data, you can just run `digits.keys()`.

Try this all out in the following DataCamp Light blocks:

script.py IPython Shell

```
# Get the keys of the `digits` data
print(digits._____)

# Print out the data
print(digits.____)

# Print out the target values
print(digits._____)

# Print out the description of the `digits` data
print(digits.DESCR)
```

Solution Run

The next thing that you can (double)check is the type of your data.

If you used `read_csv()` to import the data, you would have had a data frame that contains just the data. There wouldn't be any description component, but you would be able to resort to, for example, `head()` or `tail()` to inspect your data. In these cases, it's always wise to read up on the data description folder!

However, this tutorial assumes that you make use of the library's data and the type of the `digits` variable is not that straightforward if you're not familiar with the library. Look at the print out in the first code chunk. You'll see that `digits` actually contains `numpy` arrays!

This is already quite vital information. But how do you access these arrays?

It's straightforward, actually: you use attributes to access the relevant arrays.

to see the target values and the `DESCR` for the description, ...

But what then?

The first thing that you should know of an array is its shape. That is the number of dimensions and items that are contained within an array. The array's shape is a tuple of integers that specify the sizes of each dimension. In other words, if you have a 3d array like this `y = np.zeros((2, 3, 4))`, the shape of your array will be `(2,3,4)`.

Now let's try to see what the shape is of these three arrays that you have distinguished (the `data`, `target` and `DESCR` arrays).

Use first the `data` attribute to isolate the numpy array from the `digits` data and then use the `shape` attribute to find out more. You can do the same for the `target` and `DESCR`. There's also the `images` attribute, which is basically the data in images. You're also going to test this out.

Check up on this statement by using the `shape` attribute on the array:

script.py IPython Shell

```
# Isolate the `digits` data
digits_data = digits.data

# Inspect the shape
print(digits_data.shape)

# Isolate the target values with `target`
digits_target = digits._____

# Inspect the shape
print(digits_target._____)

# Print the number of unique labels
number_digits = len(np.unique(digits.target))
```

Solution Run

To recap: by inspecting `digits.data`, you see that there are 1797 samples and that there are 64 features. Because you have 1797 samples, you also have 1797 target values.

the digits that your model will need to recognize are numbers from 0 to 9.

Lastly, you see that the `images` data contains three dimensions: there are 1797 instances that are 8 by 8 pixels big. You can visually check that the `images` and the `data` are related by reshaping the `images` array to two dimensions: `digits.images.reshape((1797, 64))`.

But if you want to be entirely sure, better to check with

```
print(np.all(digits.images.reshape((1797,64)) == digits.data))
```

With the `numpy` method `all()`, you test whether all array elements along a given axis evaluate to `True`. In this case, you evaluate if it's true that the reshaped `images` array equals `digits.data`. You'll see that the result will be `True` in this case.

Visualize Your Data Images With `matplotlib`

Then, you can take your exploration up a notch by visualizing the images that you'll be working with. You can use one of Python's data visualization libraries, such as `matplotlib`, for this purpose:

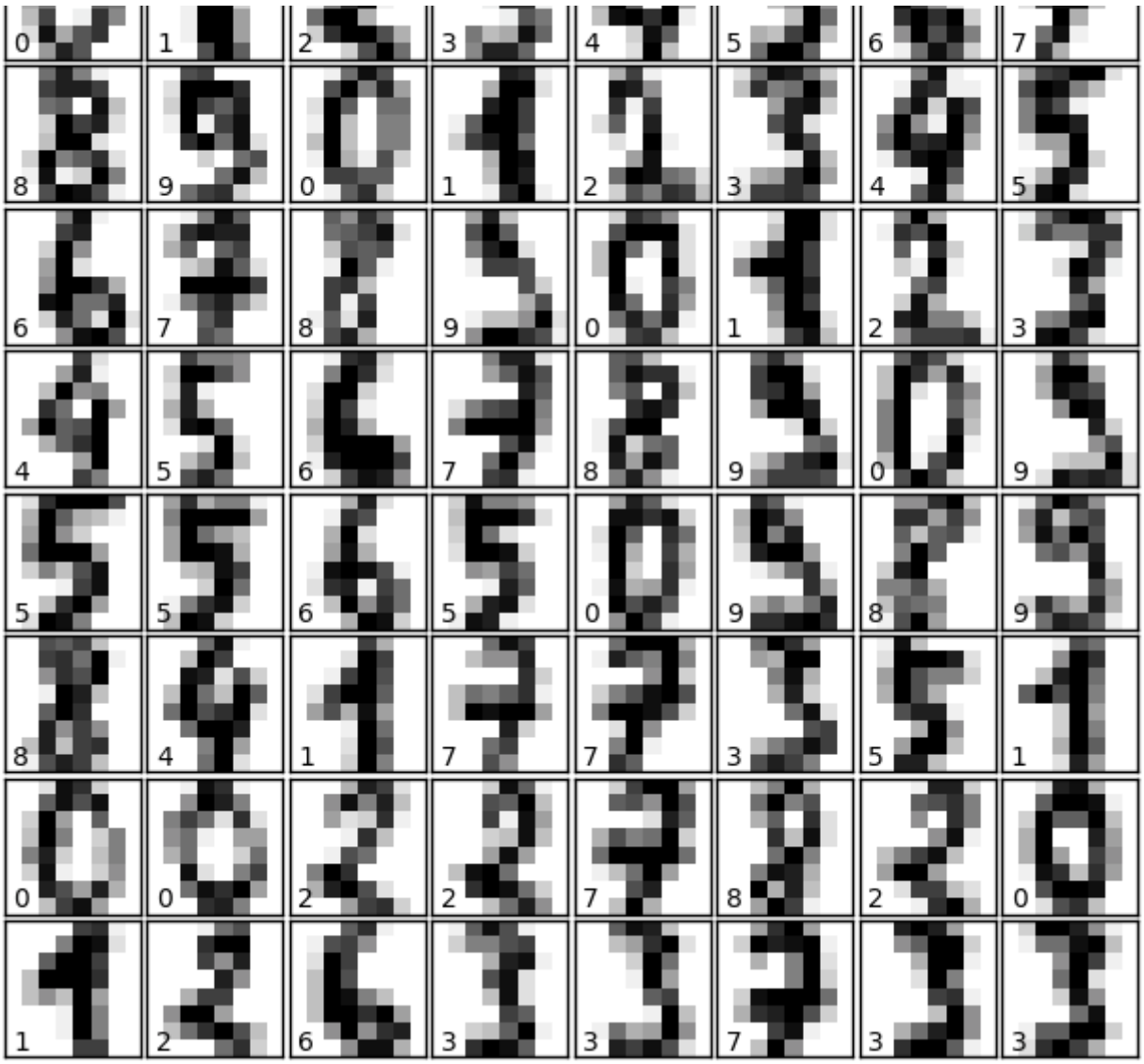
```
# Import matplotlib
import matplotlib.pyplot as plt

# Figure size (width, height) in inches
fig = plt.figure(figsize=(6, 6))

# Adjust the subplots
fig.subplots_adjust(left=0, right=1, bottom=0, top=1, hspace=0.05, wspace=0.05)

# For each of the 64 images
for i in range(64):
    # Initialize the subplots: add a subplot in the grid of 8 by 8, at the i+1-th position
    ax = fig.add_subplot(8, 8, i + 1, xticks=[], yticks=[])
    # Display an image at the i-th position
    ax.imshow(digits.images[i], cmap=plt.cm.binary, interpolation='nearest')
    # label the image with the target value
```

```
# Show the plot
plt.show()
```

The code chunk seems quite lengthy at first sight, and this might be overwhelming. But, what happens in the code chunk above is actually pretty easy once you break it down into parts:

- You import `matplotlib.pyplot`.
- Next, you set up a figure with a figure size of `6` inches wide and `6` inches long. This is your blank canvas where all the subplots with the images will appear.
- Then you go to the level of the subplots to adjust some parameters: you set the left side of the suplots of the figure to `0`, the right side of the suplots of the figure to `1`, the bottom to `0` and the top to `1`. The height of the blank space between the suplots is set at `0.005` and the width is set at `0.05`. These are merely layout adjustments.
- After that, you start filling up the figure that you have made with the help of a for loop.
- You initialize the suplots one by one, adding one at each position in the grid that is `8` by `8` images big.
- You display each time one of the images at each position in the grid. As a color map, you take binary colors, which in this case will result in black, gray values and white colors. The interpolation method that you use is `'nearest'`, which means that your data is interpolated in such a way that it isn't smooth. You can see the effect of the different interpolation methods here.
- The cherry on the pie is the addition of text to your subplots. The target labels are printed at coordinates (0,7) of each subplot, which in practice means that they will appear in the bottom-left of each of the subplots.
- Don't forget to show the plot with `plt.show()`!

In the end, you'll get to see the following:

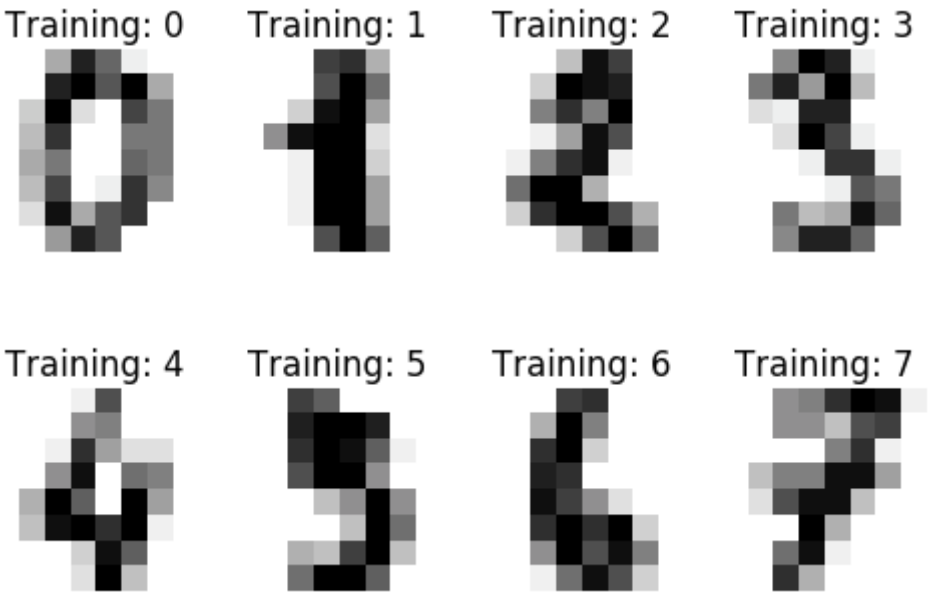


On a more simple note, you can also visualize the target labels with an image, just like this:

```
# Import matplotlib
import matplotlib.pyplot as plt

# Join the images and target labels in a list
images_and_labels = list(zip(digits.images, digits.target))

# for every element in the list
for index, (image, label) in enumerate(images_and_labels[:8]):
    # initialize a subplot of 2X4 at the i+1-th position
    plt.subplot(2, 4, index + 1)
    # Don't plot any axes
```

```
        plt.imshow(image, cmap=plt.cm.gray_r,interpolation='nearest')
        # Add a title to each subplot
        plt.title('Training: ' + str(label))

    # Show the plot
    plt.show()
```

Which will render the following visualization:



Note that in this case, after you have imported `matplotlib.pyplot`, you zip the two `numpy` arrays together and save it into a variable called `images_and_labels`. You'll see now that this list contains suples of each time an instance of `digits.images` and a corresponding `digits.target` value.

Then, you say that for the first eight elements of `images_and_labels` -note that the index starts at 0!-, you initialize subplots in a grid of 2 by 4 at each position. You turn of the plotting of the axes and you display images in all the subplots with a color map `plt.cm.gray_r` (which returns all grey colors) and the interpolation method used is `nearest`. You give a title to each subplot, and you show it.

Not too hard, huh?

Visualizing Your Data: Principal Component Analysis (PCA)

But is there no other way to visualize the data?

As the `digits` data set contains 64 features, this might prove to be a challenging task. You can imagine that it's tough to understand the structure and keep the overview of the `digits` data. In such cases, it is said that you're working with a high dimensional data set.

High dimensionality of data is a direct result of trying to describe the objects via a collection of features. Other examples of high dimensional data are, for example, financial data, climate data, neuroimaging, ...

But, as you might have gathered already, this is not always easy. In some cases, high dimensionality can be problematic, as your algorithms will need to take into account too many features. In such cases, you speak of the curse of dimensionality. Because having a lot of dimensions can also mean that your data points are far away from virtually every other point, which makes the distances between the data points uninformative.

Don't worry, though, because the curse of dimensionality is not merely a matter of counting the number of features. There are also cases in which the effective dimensionality might be much smaller than the number of the features, such as in data sets where some features are irrelevant.

In addition, you can also understand that data with only two or three dimensions are easier to grasp and can also be visualized easily.

That all explains why you're going to visualize the data with the help of one of the Dimensionality Reduction techniques, namely Principal Component Analysis (PCA). The idea in PCA is to find a linear combination of the two variables that contains most of the information. This new variable or "principal component" can replace the two original variables.

In short, it's a linear transformation method that yields the directions (principal components) that maximize the variance of the data. Remember that the variance indicates how far a set of data points lie apart. If you want to know more, go to this page.

script.py IPython Shell

```
# Create a Randomized PCA model that takes two components
randomized_pca = RandomizedPCA(n_components=2)

# Fit and transform the data to the model
reduced_data_rpca = randomized_pca.fit_transform(digits.data)

# Create a regular PCA model
pca = PCA(n_components=2)

# Fit and transform the data to the model
reduced_data_pca = pca.fit_transform(digits.data)

# Inspect the shape
reduced_data_pca.shape
```

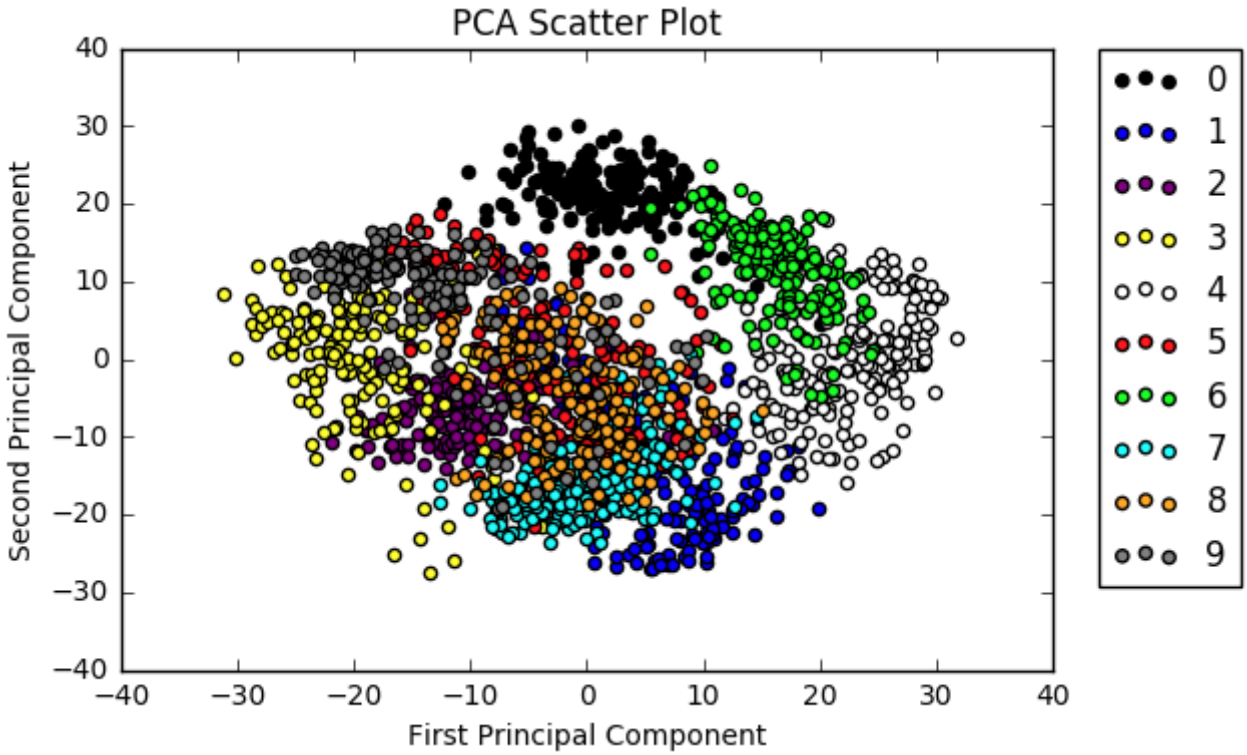
Solution Run

Tip: you have used the `RandomizedPCA()` here because it performs better when there's a high number of dimensions. Try replacing the randomized PCA model or estimator object with a regular PCA model and see what the difference is.

Note how you explicitly tell the model only to keep two components. This is to make sure that you have two-dimensional data to plot. Also, note that you don't pass the target class with the labels to the PCA transformation because you want to investigate if the PCA reveals the distribution of the different labels and if you can clearly separate the instances from each other.

You can now build a scatterplot to visualize the data:

```
colors = ['black', 'blue', 'purple', 'yellow', 'white', 'red', 'lime', 'cyan', 'orange', 'g
for i in range(len(colors)):
    x = reduced_data_rpca[:, 0][digits.target == i]
    y = reduced_data_rpca[:, 1][digits.target == i]
    plt.scatter(x, y, c=colors[i])
plt.legend(digits.target_names, bbox_to_anchor=(1.05, 1), loc=2, borderaxespad=0.)
plt.xlabel('First Principal Component')
plt.ylabel('Second Principal Component')
plt.title("PCA Scatter Plot")
plt.show()
```



Again you use `matplotlib` to visualize the data. It's useful for a quick visualization of what you're working with, but you might have to consider something a little bit fancier if you're working on making this part of your data science portfolio.

Also note that the last call to show the plot (`plt.show()`) is not necessary if you're working in Jupyter Notebook, as you'll want to put the images inline. When in doubt, you can always check out our Definitive Guide to Jupyter Notebook.

What happens in the code chunk above is the following:

1. You put your colors together in a list. Note that you list ten colors, which is equal to the number of labels that you have. This way, you make sure that your data points can be colored in according to the labels. Then, you set up a range that goes from 0 to 10. Mind you that this range is not inclusive! Remember that this is the same for indices of a list, for example.

2. You set up your `x` and `y` coordinates. You take the first or the second column of `reduced_data_rpca` , and you select only those data points for which the label equals

3. You construct the scatter plot. Fill in the `x` and `y` coordinates and assign a color to the batch that you're processing. The first run, you'll give the color `black` to all data points, the next run `blue`, ... and so on.
4. You add a legend to your scatter plot. Use the `target_names` key to get the right labels for your data points.
5. Add labels to your `x` and `y` axes that are meaningful.
6. Reveal the resulting plot.

Where To Go Now?

Now that you have even more information about your data and you have a visualization ready, it does seem a bit like the data points sort of group together, but you also see there is quite some overlap.

This might be interesting to investigate further.

Do you think that, in a case where you knew that there are 10 possible digits labels to assign to the data points, but you have no access to the labels, the observations would group or "cluster" together by some criterion in such a way that you could infer the labels?

Now, this is a research question!

In general, when you have acquired a good understanding of your data, you have to decide on the use cases that would be relevant to your data set. In other words, you think about what your data set might teach you or what you think you can learn from your data.

From there on, you can think about what kind of algorithms you would be able to apply to your data set in order to get the results that you think you can obtain.

Tip: the more familiar you are with your data, the easier it will be to assess the use cases for your specific data set. The same also holds for finding the appropriate machine

However, when you're first getting started with `scikit-learn`, you'll see that the amount of algorithms that the library contains is pretty vast and that you might still want additional help when you're assessing your data set. That's why this `scikit-learn` machine learning map will come in handy.

Note that this map does require you to have some knowledge about the algorithms that are included in the `scikit-learn` library. This, by the way, also holds some truth for taking this next step in your project: if you have no idea what is possible, it will be tough to decide on what your use case will be for the data.

As your use case was one for clustering, you can follow the path on the map towards "KMeans". You'll see the use case that you have just thought about requires you to have more than 50 samples ("check!"), to have labeled data ("check!"), to know the number of categories that you want to predict ("check!") and to have less than 10K samples ("check!").

But what exactly is the K-Means algorithm?

It is one of the simplest and widely used unsupervised learning algorithms to solve clustering problems. The procedure follows a simple and easy way to classify a given data set through a certain number of clusters that you have configured before you run the algorithm. This number of clusters is called `k`, and you select this number at random.

Then, the k-means algorithm will find the nearest cluster center for each data point and assign the data point closest to that cluster.

Once all data points have been assigned to clusters, the cluster centers will be recomputed. In other words, new cluster centers will emerge from the average of the values of the cluster data points. This process is repeated until most data points stick to the same cluster. The cluster membership should stabilize.

You can already see that, because the k-means algorithm works the way it does, the initial set of cluster centers that you give up can have a significant effect on the clusters that are eventually found. You can, of course, deal with this effect, as you will see further on.

However, before you can go into making a model for your data, you should definitely take a look into preparing your data for this purpose.

Preprocessing Your Data

As you have read in the previous section, before modeling your data, you'll do well by preparing it first. This preparation step is called "preprocessing".

Data Normalization

The first thing that we're going to do is preprocessing the data. You can standardize the `digits` data by, for example, making use of the `scale()` method:

script.py IPython Shell

```
# Import
from sklearn.preprocessing import scale

# Apply `scale()` to the `digits` data
data = _____(digits.data)
```

Solution Run

By scaling the data, you shift the distribution of each attribute to have a mean of zero and a standard deviation of one (unit variance).

Splitting Your Data Into Training And Test Sets

second is used to evaluate the learned or trained system.

In practice, the division of your data set into a test and a training sets are disjoint: the most common splitting choice is to take 2/3 of your original data set as the training set, while the 1/3 that remains will compose the test set.

You will try to do this also here. You see in the code chunk below that this 'traditional' splitting choice is respected: in the arguments of the `train_test_split()` method, you clearly see that the `test_size` is set to `0.25`.

You'll also note that the argument `random_state` has the value `42` assigned to it. With this argument, you can guarantee that your split will always be the same. That is particularly handy if you want reproducible results.

script.py IPython Shell

```
# Import `train_test_split`
from sklearn.cross_validation import _________________

# Split the `digits` data into training and test sets
X_train, X_test, y_train, y_test, images_train, images_test = train_test_split
(data, digits.target, digits.images, test_size=0.25, random_state=42)
```

Solution Run

After you have split up your data set into train and test sets, you can quickly inspect the numbers before you go and model the data:

script.py IPython Shell

```
# Number of training features
n_samples, n_features = X_train.shape

# Print out `n_samples`
print(_________)

# Print out `n_features`
print(__________)

```

```
# Inspect `y_train`
print(len(       ))
```

Solution Run

You'll see that the training set `X_train` now contains 1347 samples, which is precisely 2/3d of the samples that the original data set contained, and 64 features, which hasn't changed. The `y_train` training set also contains 2/3d of the labels of the original data set. This means that the test sets `X_test` and `y_test` contains 450 samples.

Clustering The `digits` Data

After all these preparation steps, you have made sure that all your known (training) data is stored. No actual model or learning was performed up until this moment.

Now, it's finally time to find those clusters of your training set. Use `KMeans()` from the `cluster` module to set up your model. You'll see that there are three arguments that are passed to this method: `init`, `n_clusters` and the `random_state`.

You might still remember this last argument from before when you split the data into training and test sets. This argument basically guaranteed that you got reproducible results.

script.py IPython Shell

```
# Import the `cluster` module
from sklearn import ________

# Create the KMeans model
clf = cluster.KMeans(init='k-means++', n_clusters=10, random_state=42)

# Fit the training data `X_train`to the model
clf.fit(________)
```

Solution Run

leave it out if you want. Try it out in the DataCamp Light chunk above!

Next, you also see that the `n_clusters` argument is set to `10`. This number not only indicates the number of clusters or groups you want your data to form, but also the number of centroids to generate. Remember that a cluster centroid is the middle of a cluster.

Do you also still remember how the previous section described this as one of the possible disadvantages of the K-Means algorithm?

That is that the initial set of cluster centers that you give up can have a significant effect on the clusters that are eventually found?

Usually, you try to deal with this effect by trying several initial sets in multiple runs and by selecting the set of clusters with the minimum sum of the squared errors (SSE). In other words, you want to minimize the distance of each point in the cluster to the mean or centroid of that cluster.

By adding the `n-init` argument to `KMeans()`, you can determine how many different centroid configurations the algorithm will try.

Note again that you don't want to insert the test labels when you fit the model to your data: these will be used to see if your model is good at predicting the actual classes of your instances!

You can also visualize the images that make up the cluster centers as follows:

```
# Import matplotlib
import matplotlib.pyplot as plt

# Figure size in inches
fig = plt.figure(figsize=(8, 3))

# Add title
fig.suptitle('Cluster Center Images', fontsize=14, fontweight='bold')

# For all labels (0-9)
```

```
    ax = fig.add_subplot(2, 5, 1 + i)
    # Display images
    ax.imshow(clf.cluster_centers_[i].reshape((8, 8)), cmap=plt.cm.binary)
    # Don't show the axes
    plt.axis('off')

# Show the plot
plt.show()
```

If you want to see another example that visualizes the data clusters and their centers, go here.

The next step is to predict the labels of the test set:

script.py IPython Shell

```
# Predict the labels for `X_test`
y_pred=clf.predict(X_test)

# Print out the first 100 instances of `y_pred`
print(y_pred[:100])

# Print out the first 100 instances of `y_test`
print(y_test[:100])

# Study the shape of the cluster centers
clf.cluster_centers_._____
```

In the code chunk above, you predict the values for the test set, which contains 450 samples. You store the result in `y_pred`. You also print out the first 100 instances of `y_pred` and `y_test`, and you immediately see some results.

In addition, you can study the shape of the cluster centers: you immediately see that there are 10 clusters with each 64 features.

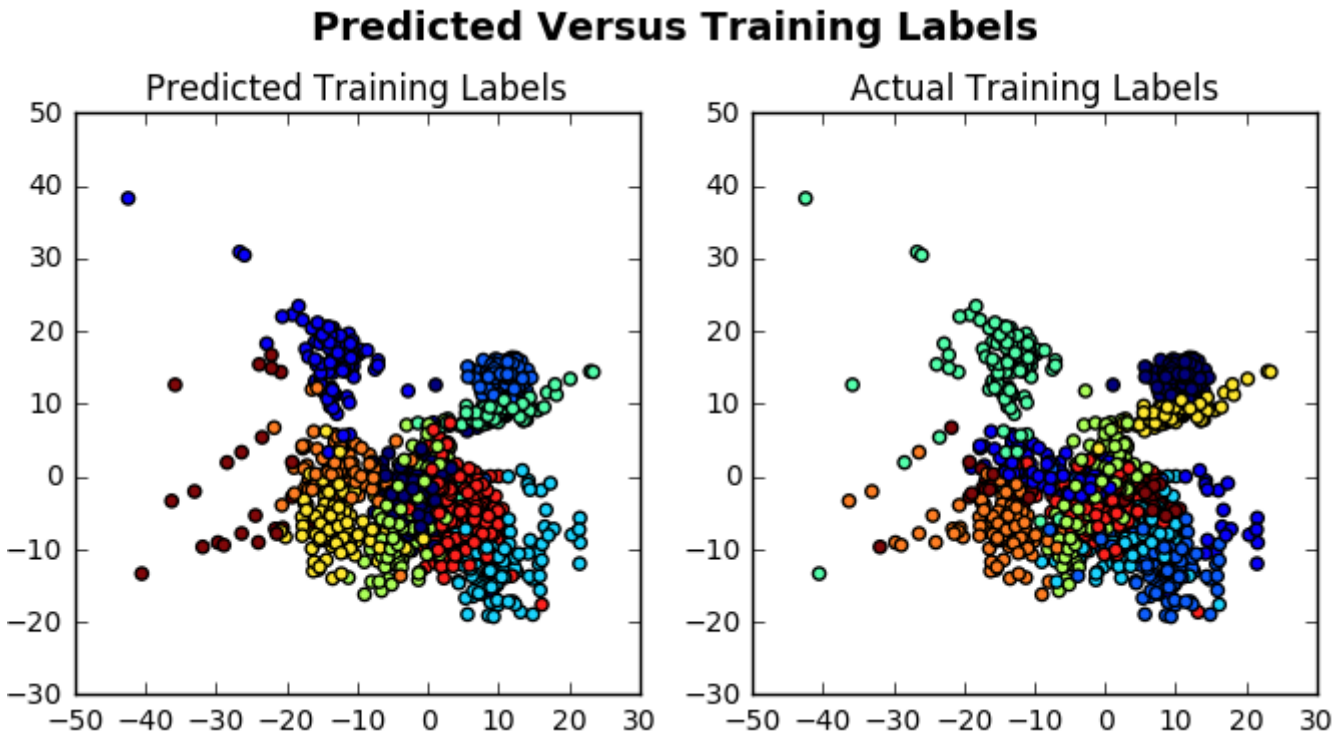
But this doesn't tell you much because we set the number of clusters to 10 and you already knew that there were 64 features.

Maybe a visualization would be more helpful.

Let's visualize the predicted labels:

```
# Import `Isomap()`
from sklearn.manifold import Isomap

# Create an isomap and fit the `digits` data to it
X_iso = Isomap(n_neighbors=10).fit_transform(X_train)

# Compute cluster centers and predict cluster index for each sample
clusters = clf.fit_predict(X_train)

# Create a plot with subplots in a grid of 1X2
fig, ax = plt.subplots(1, 2, figsize=(8, 4))

# Adjust layout
fig.suptitle('Predicted Versus Training Labels', fontsize=14, fontweight='bold')
fig.subplots_adjust(top=0.85)

# Add scatterplots to the subplots
ax[0].scatter(X_iso[:, 0], X_iso[:, 1], c=clusters)
ax[0].set_title('Predicted Training Labels')
ax[1].scatter(X_iso[:, 0], X_iso[:, 1], c=y_train)
ax[1].set_title('Actual Training Labels')
```

You use `Isomap()` as a way to reduce the dimensions of your high-dimensional data set `digits`. The difference with the PCA method is that the Isomap is a non-linear reduction method.

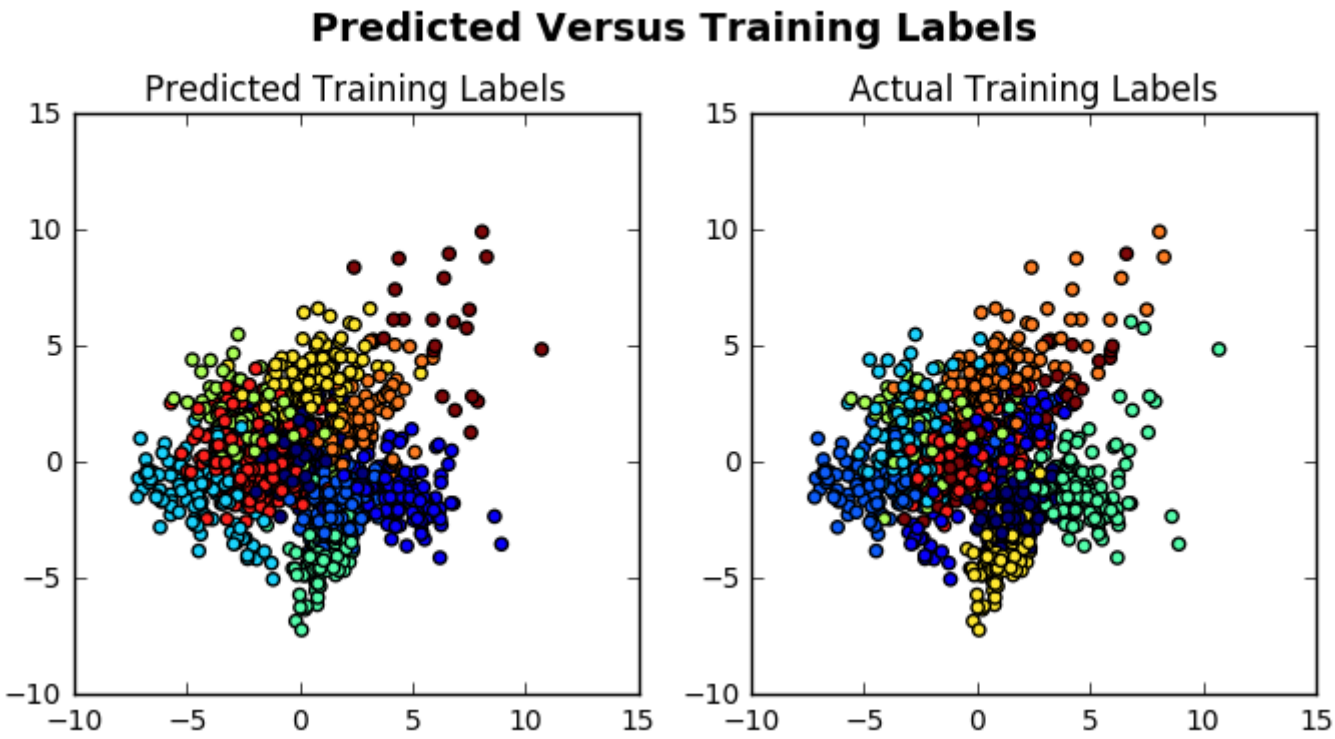


Tip: run the code from above again, but use the PCA reduction method instead of the Isomap to study the effect of reduction methods yourself.

You will find the solution here:

```
# Import `PCA()`
from sklearn.decomposition import PCA

# Model and fit the `digits` data to the PCA model
X_pca = PCA(n_components=2).fit_transform(X_train)

# Compute cluster centers and predict cluster index for each sample
clusters = clf.fit_predict(X_train)

# Create a plot with subplots in a grid of 1X2
```

```
# Adjust layout
fig.suptitle('Predicted Versus Training Labels', fontsize=14, fontweight='bold')
fig.subplots_adjust(top=0.85)

# Add scatterplots to the subplots
ax[0].scatter(X_pca[:, 0], X_pca[:, 1], c=clusters)
ax[0].set_title('Predicted Training Labels')
ax[1].scatter(X_pca[:, 0], X_pca[:, 1], c=y_train)
ax[1].set_title('Actual Training Labels')

# Show the plots
plt.show()
```



At first sight, the visualization doesn't seem to indicate that the model works well.

But this needs some further investigation.

Evaluation of Your Clustering Model

And this need for further investigation brings you to the next essential step, which is the evaluation of your model's performance. In other words, you want to analyze the degree

Let's print out a confusion matrix:

script.py IPython Shell

```
# Import `metrics` from `sklearn`
from sklearn import _______

# Print out the confusion matrix with `confusion_matrix()`
print(metrics.confusion_matrix(y_test, y_pred))
```

Solution Run

At first sight, the results seem to confirm our first thoughts that you gathered from the visualizations. Only the digit `5` was classified correctly in 41 cases. Also, the digit `8` was classified correctly in 11 instances. But this is not really a success.

You might need to know a bit more about the results than just the confusion matrix.

Let's try to figure out something more about the quality of the clusters by applying different cluster quality metrics. That way, you can judge the goodness of fit of the cluster labels to the correct labels.

script.py IPython Shell

```
from sklearn.metrics import homogeneity_score, completeness_score,
v_measure_score, adjusted_rand_score, adjusted_mutual_info_score,
silhouette_score
print('% 9s' % 'inertia    homo   compl  v-meas     ARI AMI  silhouette')
print('%i   %.3f   %.3f   %.3f   %.3f   %.3f    %.3f'
          %(clf.inertia_,
        homogeneity_score(y_test, y_pred),
        completeness_score(y_test, y_pred),
        v_measure_score(y_test, y_pred),
        adjusted_rand_score(y_test, y_pred),
        adjusted_mutual_info_score(y_test, y_pred),
        silhouette_score(X_test, y_pred, metric='euclidean')))
```

Run

You'll see that there are quite some metrics to consider:

- The homogeneity score tells you to what extent all of the clusters contain only data points which are members of a single class.
- The completeness score measures the extent to which all of the data points that are members of a given class are also elements of the same cluster.
- The V-measure score is the harmonic mean between homogeneity and completeness.
- The adjusted Rand score measures the similarity between two clusterings and considers all pairs of samples and counting pairs that are assigned in the same or different clusters in the predicted and true clusterings.
- The Adjusted Mutual Info (AMI) score is used to compare clusters. It measures the similarity between the data points that are in the clusterings, accounting for chance groupings and takes a maximum value of 1 when clusterings are equivalent.
- The silhouette score measures how similar an object is to its own cluster compared to other clusters. The silhouette scores range from -1 to 1, where a higher value indicates that the object is better matched to its own cluster and worse matched to neighboring clusters. If many points have a high value, the clustering configuration is good.

You clearly see that these scores aren't fantastic: for example, you see that the value for the silhouette score is close to 0, which indicates that the sample is on or very close to the decision boundary between two neighboring clusters. This could indicate that the samples could have been assigned to the wrong cluster.

Also, the ARI measure seems to indicate that not all data points in a given cluster are similar and the completeness score tells you that there are definitely data points that weren't put in the right cluster.

Clearly, you should consider another estimator to predict the labels for the `digits` data.

Trying Out Another Model: Support Vector Machines

you knowing the labels. And indeed, you just used the training data and not the target values to build your KMeans model.

Let's assume that you depart from the case where you use both the `digits` training data and the corresponding target values to build your model.

If you follow the algorithm map, you'll see that the first model that you meet is the linear SVC. Let's apply this now to the `digits` data:

script.py IPython Shell

```
# Import `train_test_split`
from sklearn.cross_validation import train_test_split

# Split the data into training and test sets
X_train, X_test, y_train, y_test, images_train, images_test =
train_test_split(digits.data, digits.target, digits.images, test_size=0.25,
random_state=42)

# Import the `svm` model
from sklearn import svm

# Create the SVC model
svc_model = svm.SVC(gamma=0.001, C=100., kernel='linear')

```

Solution Run

You see here that you make use of `X_train` and `y_train` to fit the data to the SVC model. This is clearly different from clustering. Note also that in this example, you set the value of `gamma` manually. It is possible to automatically find good values for the parameters by using tools such as grid search and cross-validation.

Even though this is not the focus of this tutorial, you will see how you could have gone about this if you would have made use of grid search to adjust your parameters. You would have done something like the following:

script.py IPython Shell

```
# Split the `digits` data into two equal sets
X_train, X_test, y_train, y_test = train_test_split(digits.data, digits
.target, test_size=0.5, random_state=0)

# Import GridSearchCV
from sklearn.grid_search import GridSearchCV

```

```
    {'C': [1, 10, 100, 1000], 'gamma': [0.001, 0.0001], 'kernel': ['rbf']},
]

# Create a classifier with the parameter candidates
```

Run

Next, you use the classifier with the classifier and parameter candidates that you have just created to apply it to the second part of your data set. Next, you also train a new classifier using the best parameters found by the grid search. You score the result to see if the best parameters that were found in the grid search are actually working.

script.py IPython Shell

```
# Apply the classifier to the test data, and view the accuracy score
clf.score(X_test, y_test)

# Train and score a new classifier with the grid search parameters
svm.SVC(C=10, kernel='rbf', gamma=0.001).fit(X_train, y_train).score(X_test,
y_test)
```

Run

The parameters indeed work well!

Now, what does this new knowledge tell you about the SVC classifier that you had modeled before you had done the grid search?

Let's back up to the model that you had made before.

You see that in the SVM classifier, the penalty parameter `C` of the error term is specified at `100.` . Lastly, you see that the kernel has been explicitly specified as a `linear` one. The `kernel` argument specifies the kernel type that you're going to use in the algorithm and by default, this is `rbf` . In other cases, you can specify others such as `linear` , `poly` , ...

A kernel is a similarity function, which is used to compute the similarity between the training data points. When you provide a kernel to an algorithm, together with the training data and the labels, you will get a classifier, as is the case here. You will have trained a model that assigns new unseen objects into a particular category. For the SVM, you will typically try to divide your data points linearly.

However, the grid search tells you that an `rbf` kernel would've worked better. The penalty parameter and the gamma were specified correctly.

Tip: try out the classifier with an `rbf` kernel.

For now, let's say you continue with a linear kernel and predict the values for the test set:

script.py IPython Shell

```
# Predict the label of `X_test`
print(svc_model.predict(_____))

# Print `y_test` to check the results
print(_____)
```

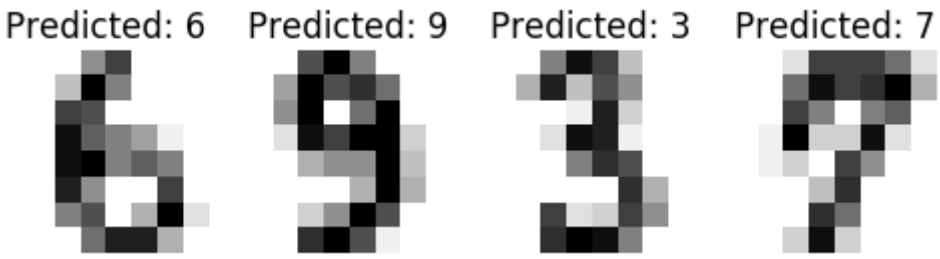
Solution Run

You can also visualize the images and their predicted labels:

```
# Import matplotlib
import matplotlib.pyplot as plt

# Assign the predicted values to `predicted`
predicted = svc_model.predict(X_test)

# Zip together the `images_test` and `predicted` values in `images_and_predictions`
images_and_predictions = list(zip(images_test, predicted))
```

```
    # Initialize subplots in a grid of 1 by 4 at positions i+1
    plt.subplot(1, 4, index + 1)
    # Don't show axes
    plt.axis('off')
    # Display images in all subplots in the grid
    plt.imshow(image, cmap=plt.cm.gray_r, interpolation='nearest')
    # Add a title to the plot
    plt.title('Predicted: ' + str(prediction))

# Show the plot
plt.show()
```

This plot is very similar to the plot that you made when you were exploring the data:



Only this time, you zip together the images and the predicted values, and you only take the first 4 elements of `images_and_predictions` .

But now the biggest question: how does this model perform?

script.py IPython Shell

```
# Import `metrics`
from sklearn import metrics

# Print the classification report of `y_test` and `predicted`
print(metrics.classification_report(_____, _________))

# Print the confusion matrix of `y_test` and `predicted`
print(metrics.confusion_matrix(_____, _________))
```

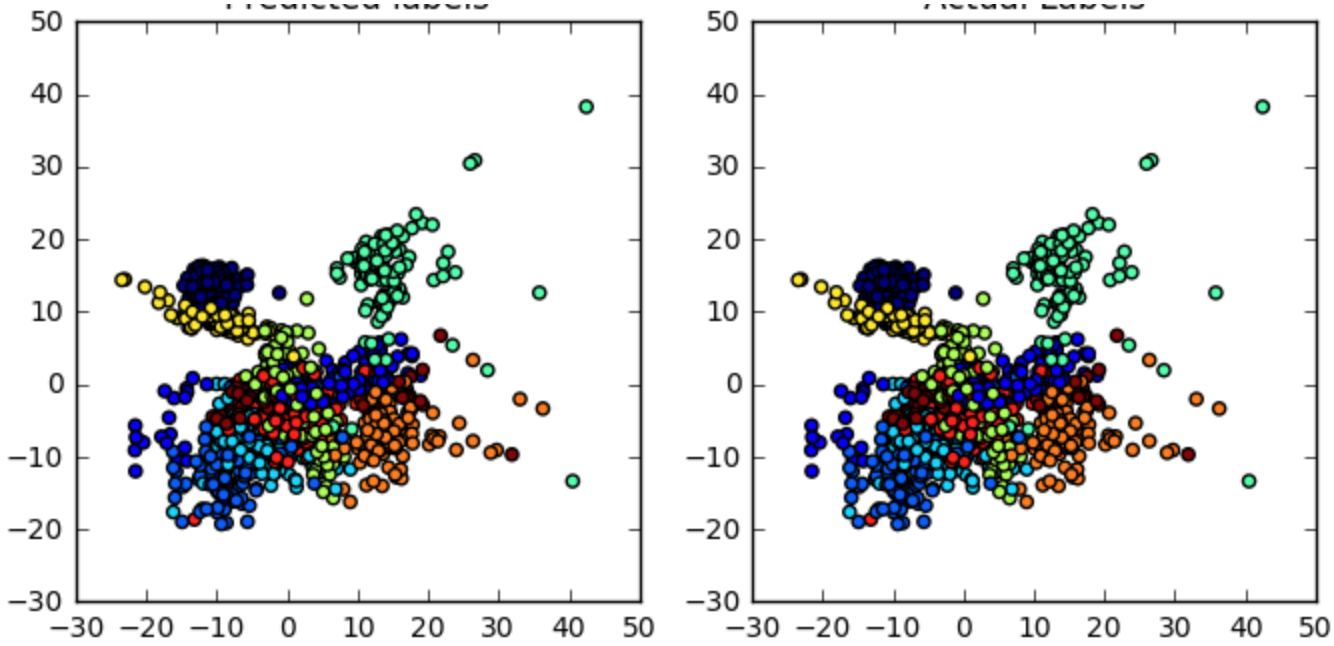
Solution Run

You clearly see that this model performs a whole lot better than the clustering model that you used earlier.

You can also see it when you visualize the predicted and the actual labels with the help of `Isomap()`:

```
# Import `Isomap()`
from sklearn.manifold import Isomap

# Create an isomap and fit the `digits` data to it
X_iso = Isomap(n_neighbors=10).fit_transform(X_train)

# Compute cluster centers and predict cluster index for each sample
predicted = svc_model.predict(X_train)

# Create a plot with subplots in a grid of 1X2
fig, ax = plt.subplots(1, 2, figsize=(8, 4))

# Adjust the layout
fig.subplots_adjust(top=0.85)

# Add scatterplots to the subplots
ax[0].scatter(X_iso[:, 0], X_iso[:, 1], c=predicted)
ax[0].set_title('Predicted labels')
ax[1].scatter(X_iso[:, 0], X_iso[:, 1], c=y_train)
ax[1].set_title('Actual Labels')



# Add title
fig.suptitle('Predicted versus actual labels', fontsize=14, fontweight='bold')

# Show the plot
plt.show()
```

This will give you the following scatterplots:



You'll see that this visualization confirms your classification report, which is excellent news. :)

What's Next?

Digit Recognition in Natural Images

Congratulations, you have reached the end of this scikit-learn tutorial, which was meant to introduce you to Python machine learning! Now it's your turn.

Firstly, make sure you get a hold of DataCamp's `scikit-learn` cheat sheet.

Next, start your own digit recognition project with different data. One dataset that you can already use is the MNIST data, which you can download here.

The steps that you can take are very similar to the ones that you have gone through with this tutorial, but if you still feel that you can use some help, you should check out this page, which works with the MNIST data and applies the KMeans algorithm.

Working with the `digits` dataset was the first step in classifying characters with `scikit-learn`. If you're done with this, you might consider trying out an even more challenging problem, namely, classifying alphanumeric characters in natural images.

case letters of the English alphabet. You can download the dataset here.

Data Visualization and pandas

Whether you're going to start with the projects that have been mentioned above or not, this is definitely not the end of your journey of data science with Python. If you choose not to widen your view just yet, consider deepening your data visualization and data manipulation knowledge.

Don't miss out on our Interactive Data Visualization with Bokeh course to make sure you can impress your peers with a stunning data science portfolio or our pandas Foundation course, to learn more about working with data frames in Python.

MUST READ MACHINE LEARNING +4

Detecting Fake News with Scikit-Learn

Katharine Jarmul
August 24th, 2017

COMMENTS

vivekvscool
23/02/2018 11:33 AM
Excellent tutorial! Learned a ton and explanation is understandable for someone new to machine learning.

▲ 4

Karlijn Willems
23/02/2018 02:42 PM
Hi vivekvscool - Great to hear! Happy that you learned a lot :)

▲ 2

---

hawkra
01/03/2018 12:05 PM
I am very unclear on the section

```
for i in range(len(colors)):

        x = reduced_data_rpca[:, 0][digits.target == i]

        y = reduced_data_rpca[:, 1][digits.target == i]

        plt.scatter(x, y, c=colors[i])
```

▲ 1

Karlijn Willems
01/03/2018 01:04 PM

Hi hawkra - what happens is that you go through the list of colors one by one. Per color, you will dynamically construct `x` and `y` arrays, which you can then pass to `plt.scatter()`. You construct these `x` and `y` arrays from `reduced_data_rpca` has a shape of `(1797, 2)` . Here's an excerpt of what that last variable looks like:

```
array([[ -1.25946584, 21.27488565],

[ 7.9576116 , -20.76869756],

[ 6.99192206, -9.95598915],

...,

[ 10.80128426, -6.96025179],

[ -4.87210311, 12.42395303],

[ -0.34438951,  6.36554784]])
```

From this array, you will first select all elements at position [0] within the array for `x` and then all elements at position [1] for `y`:

```
x = reduced_data_rpca[:, 0]

y = reduced_data_rpca[:, 1]
```

gives you, for `x`:

```
-4.88722556, -0.34254567])
```

and for `y`, you should see:

```
array([ 21.29635137, -20.77095636, -9.92995725, ..., -6.94087344,


12.41295392,  6.36962795])
```

So far so good.

However, if you would like to make a scatterplot, you would want to visualize it in such a way to see whether or not the data has natural clusters in it, according to the target values. That's why, from the `x` and `y` that you have just constructed, you need to isolate the correct data that belongs to each target (written digit). That's why you will want to add `[digits.target == i]` to your code. As you go through the `colors` list, you `i`'s value will change from 0 to 9 (exactly the number of written digits or target variables you have!).

That means that, in the first pass of your loop, the code will "look" this, as the `i` will be at `0`:

```
x = reduced_data_rpca[:, 0][digits.target == 0]


y = reduced_data_rpca[:, 1][digits.target == 0]
```

In other words, you go through `x` and `y` and you only select those values that have a target value of 0 (that are actually labelled as "0"). You construct them, and you plot them. Then you go through the loop, and the second pass will look like this:

```
x = reduced_data_rpca[:, 0][digits.target == 1]


y = reduced_data_rpca[:, 1][digits.target == 1]
```

And so on.

whether or not there are some type of clusters when you finish plotting.

▲ 3

hawkra
02/03/2018 01:38 AM
Very detailed and clear explanation, truly appreciate your time!

▲ 1

hawkra
02/03/2018 02:00 AM
So would

```
x = reduced_data_rpca[:, 0][digits.target == 0]
```

be equivalent to:

```
x = reduced_data_rpca[:, 0][[0 1 2 ... 8 9 8] == 0]
```

And would this mean that the number '10.78897583' (contained within the reduced_data_rpca[:, 0])would not be included in the representation as it is out of the range ?

▲ 1

Karlijn Willems
02/03/2018 02:18 PM
Hi Hawkra -

```
x = reduced_data_rpca[:, 0][digits.target == 0]
```

Just means that you're going to take the data that is the result of `reduced_data_rpca`, that is,

```
array([ -1.24792501,  7.96053635,  7.01873296, ..., 10.81940109, -4.88722556,
```

And from that data, you're only going to select the data points that have a target label `0`, for example, in the first pass of your loop. In the second pass of the for loop, you're going to go through the above data again and retain only those data points that have a target

▲ 2

tapannayyar
08/07/2018 04:24 PM
[:,x] This signifies that you have selected all the rows and the xth column of the array.

▲ 1

---

hawkra
02/03/2018 01:34 AM
In the following section:

> *You'll see that the training set `X_train` now contains 1347 samples, which is exactly 2/3d of the samples that the original data set contained, and 64 features, which hasn't changed. The `y_train` training set also contains 2/3d of the labels of the original data set. This means that the test sets `X_train` and `y_train` contain 450 samples.*

for the X_train and y_train containing 450 samples, did you mean the X_test/y_test?

▲ 1

Karlijn Willems
02/03/2018 02:22 PM
Hi Hawkra - Yes, you're correct here. I have made the necessary changes within the tutorial :) Thanks for pointing that one out!

▲ 1

Hansee Han
15/03/2018 10:56 PM
Was following : randomized_pca = RandomizedPCA(n_components=2)

> *Class RandomizedPCA is deprecated; RandomizedPCA was deprecated in 0.18 and will be removed in 0.20. Use PCA(svd_solver='randomized') instead. The new implementation DOES NOT store whiten ``components_``. Apply transform to get them.*

▲ 1

A jupyter notebook of the same can help!

For some reason, i tried recreating the same in my local machine but it didn't work out!

▲ 2

Manohar Sri
05/04/2018 10:14 PM
@**Karlijn Willems :**

I tried recreating the same in my local machine, it doesn't seem to work?

Any thoughts, please.

▲ 1

Karlijn Willems
06/04/2018 01:03 PM
Hi Manohar. It's particularly hard to offer any assistance if you don't specify exactly what problem you're facing in recreating the tutorial on your local machine. Any error messages or difficulties that you might find in doing so and that you could describe here would be of great help.

▲ 1

Isaac Benchetrit
22/04/2018 03:16 PM
As your use case was one for clustering, you can follow the path on the map towards "KMeans". You'll see the use case that you have just thought about requires you to have more than 50 samples ("check!"), ***to have labeled data*** *("check!"), to know the number of categories that you want to predict ("check!") and to have less than 10K samples ("check!").*

KMeans works without labeled data .

▲ 4

Isaac Benchetrit
22/04/2018 04:24 PM

`y_train`training set also contains 2/3d of the labels of the original data set. *This means that the test sets `X_test` and `y_test` contain 450 samples.*

1347 samples correspond to approximately 3/4 of the 1797 initial samples : the test_size = 0.25 so the train_size = 0.75=3/4 not 2/3.

The same for the y_train size...

▲ 6

Luis Messy
09/03/2020 12:51 AM
send text messages online .

▲ 1

---

ranjanint07
29/05/2018 08:14 PM
Hi,

Excellent article for beginner to ML.

I have one question,

We can visualize the training and test set data using various python libraries.But if i have to show the attributes of a particular data point say for example i have an Employee table with columns 'Employee Name', 'Experience' and 'Salary' and i use may be logistic regression to find out what would be the salary when a new employee joins.

I want to present this in a board room meeting.I got the data point distributions and all using logistic regression but i need to show them the attributes 'Employee Name' ,'Experience' and the 'salary' that is predicted when i hover my mouse on any data point.

Is there a way to show these attribute details anyhow?

▲ 1

Luis Messy
06/03/2020 07:50 PM
yes and also you can try mp3juice for mp3 .

Jay Pagnis
12/06/2018 10:35 AM
The link to the **kaggle** website in the third para of the section at the top **"Loading your dataset"** is incorrect.

▲ 2

jack morris
30/06/2019 12:43 PM
that link worked great for me, seems correct.

▲ 1

Michael Steven
20/08/2019 07:23 PM
This website works verry good, you can really get free eshop codes

▲ 1

Shubhangi Agrawal
16/06/2018 01:20 AM
Hi Karlijn,

Thanks so much for the great tutorial! I had a question about the preprocessing stage where you normalized the data. Why wasn't the data normalized before applying the PCA? I assume, depending on the data set, that if the data is on a different scale for different features, the PCA may be heavily biased towards towards features that are on a bigger scale? In this case, is it because we can assume that the data for each of the 64 features (assuming each feature corresponds to pixel intensity) in the digits.data array falls in the same scale?

▲ 1

Sanjay Sane
30/07/2018 11:11 PM
Very Nicely explained steps

▲ 1

Whom Ever
08/09/2019 01:13 PM
I have learnt a lot from this lesson, I got a bit confused until I found this link but thanks.

▲ 1

morrison white
10/09/2019 07:41 PM

▲ 1

Jonathan Kaija
12/08/2018 01:20 PM

I think the train test split you used is 3/4 and I/4 respectively since test_size = 0.25.

i.e. for the train_test_split() module, and thus it is not 2/3 and 1/3

▲ 2

morrison white
19/09/2019 01:12 AM

very detail and clear explanation. link

▲ 1

Luis Messy
06/03/2020 02:08 AM

Thank You, You should checkout paintball shop .

▲ 1

Hajar Merizak
15/11/2018 10:21 PM

didn't understand the meaning of target an keys attribute ?

▲ 2

vidyad sagar
17/03/2019 02:21 AM

target variable means outcome variable or dependent variable

▲ 1

morrison white
10/09/2019 07:45 PM

check the link for better explanation.

▲ 1

Luis Messy
12/03/2020 09:37 PM

seo amsterdam no cure no pay .

▲ 1

very nicely written. thank you so much to share this .

▲ 1

Luis Messy
06/03/2020 02:10 AM
Yes Nice Post .

▲ 1

Gabriel Voiculescu
01/05/2019 01:14 PM
Very bad explained ... :(

▲ 1

Whom Ever
08/09/2019 01:16 PM
You are very right. I had to focus on my best archero hack to focus on better things with my time. Still a great tutorial BTW.

▲ 1

Zhao xudong
11/06/2019 02:53 PM
randomized_pca = RandomizedPCA(n_components=2)

hello ! teacher. In this code! I find a question . this function is only can run in 0.17, and i can't run it in 0.21.2

and my solution is

```

from sklearn.decomposition import PCA

randomized_pca = PCA(n_components=2,svd_solver='randomized')

```

▲ 2

Ebrahim Kadwa
26/06/2019 03:53 PM

changes to the following with new scikit-learn

```
from sklearn.model_selection import train_test_split
```

▲ 1

Ebrahim Kadwa
26/06/2019 04:21 PM
GridSearchCV also in model_selection

▲ 1

jack morris
17/08/2019 05:18 PM
thanks this spin worked for me.

▲ 1

Vivekanandhan Rajan
20/09/2019 06:24 PM
can anyone explain for which type of dataset ,which algorithm is more suitable.?

▲ 1

ncsun ncsj
16/10/2019 03:33 PM
The important thing in the digital marketing it is very easy to adapt and connect with the target audience worldwide. There are multiple chapters under digital marketing category which is coming with different modules. A few prime modules details have been briefly given below for the reference. **http://amplifytexas.com**

▲ 1

Key media
24/10/2019 06:16 PM
Valuable information. Fortunate me I found your site by accident,
and I am shocked why this twist of fate did not happened
earlier! I bookmarked it. https://www.keyanalyzer.com/best-youtube-alternatives/

▲ 1

ncsun ncsj
16/10/2019 03:33 PM

in addition to answer this question "how do you get far more leads intended for my small business? " -- which work better purpose connected with any advertising and marketing.
http://512Review.com

▲ 1

Tao Mei
27/10/2019 03:01 AM
Great tutorial, helped me on my project! Thanks.

▲ 1

john albert
03/02/2020 09:05 PM
ya you should also try pop slots free chips

▲ 1

Willie Picke
18/11/2019 03:29 PM
Really impressive! the situationist have helped me understand it better. Thanks.

▲ 1

john meo
20/02/2020 03:23 AM
dogs are best human friend, read on why dogs

▲ 1

Pieter Wessels
29/11/2019 09:54 PM
I found this tutorial very useful, thank you.

However, I think there is a significant flaw in it, particularly in the interpretation of the CONFUSION MATRIX. The author incorrectly assumes that the first cluster centroid relates to the first label (in this case the labels are 0,1,2,3...9 - so the author assumes, for example, that the fifth cluster centroid IS the digit "5") . This is not the case. The fact that (a) we're specifically selected 10 centroids and (b) that the centroids just happen to be called 0,1,2,3...9 does not mean that centroid 0 equates to digit "0".

This is clearly demonstrated by looking at the Cluster Centroid Images - where it appears (and again the labels are not provided so the deduction is an assumption) that the second centroid is (vaguely) "4", the third centroid is vaguely "0", etc.

exception). I'm willing to guess based on the Cluster Center Images and these results, that these columns probably represent digits "0" and "6"

Rather than use k-Means clustering (without investigating the results of the clusters), k-NN (Nearest Neighbours) works very well ,so that using the default of k=5, for example, produces an accuracy of 0.97

▲ 1

jousha Jackson
02/03/2020 01:42 PM
true that pop slots free chips

▲ 1

Alina Nic
05/12/2019 12:07 PM
Yes This is Very Helpful to Get coin master free spins and thanks for this

▲ 1

Veeru Gandhad
29/01/2020 08:54 PM
Hello

i want to build a data linage kind of visulization for my scheduled batch which is dependent flow from one job to another or multiple and vice versa

using python , could some one quickly help me which algorithm or any solution to build the visualization for that

▲ 1

Rex Dantani
15/02/2020 08:55 AM
Hi Karlijn,

Thanks you so much for this tutorial. I learnt a lot. Can you please help me in following error. This is the final result. My predicted values are different than the image shown there. It's strange but very excited to learn something new. First 4 elements are 2 8 2 6 in my results and not 6 9 3 6

Why it's showing different image than predicted values??

▲ 1

john meo
20/02/2020 03:19 AM

u know hypnotherapy is best therapy to lose weight, you should try it.

▲ 1

jousha Jackson
23/02/2020 04:41 PM

Ascension or spirituality awakening appears to be a choice either consciously or subconsciously at a soul level to expand

▲ 1

jousha Jackson
25/02/2020 04:01 PM

trading is best way to earn money online, you should try it

▲ 1

Chiranji Bera
12/03/2020 11:36 AM

Yes, Correct! but much Skills and Experience required I guess to earn money from this profession. I prefer Local Franchise business to earn decent money. Franchise Lo is a blog where you can find awesome franchise business opportunities.

▲ 1

jousha Jackson
01/03/2020 12:25 AM

Bill Hunt Public Relations is public relation consultant in UK.

▲ 1

jousha Jackson
01/03/2020 07:21 PM

You should check photos by auschwitz-birkenau