Anomaly Detection with K-Means Clustering

Aug 9, 2015

This post is a static reproduction of an IPython notebook prepared for a machine learning workshop given to the Systems group at Sanger, which aimed to give an introduction to machine learning techniques in a context relevant to systems administration.

The workshop was split into two halves: supervised learning and unsupervised learning. The supervised portion was produced by Elena Chatzimichali; the unsupervised half is what you see below. We take a look at a simple example of *k*-means clustering for anomaly detection in time series data. This example is based on Chapter 4, *More Complex, Adaptive Models* from Practical Machine Learning by Ted Dunning and Ellen Friedman.

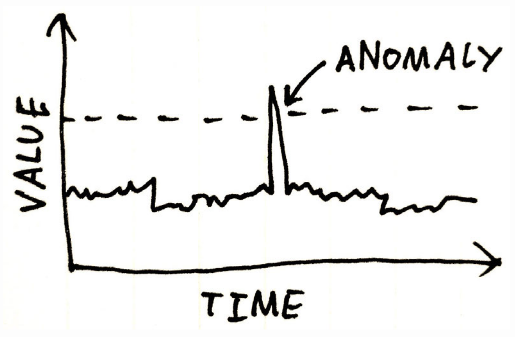
The source for the notebook and listings of the full code developed in the workbook are available on GitHub at https://github.com/mrahtz/sanger-machine-learning-workshop .

Update: Majid al-Dosari (in the comments below) and Eamonn Keogh point out that there may be issues with the approach described here for the reasons outlined in Clustering of Time Series Subsequences is Meaningless . This material still serves as an introduction to unsupervised learning and clustering, but beware in using it for anomaly detection in practice.

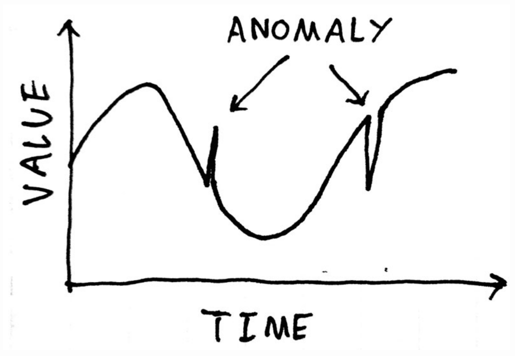
Simple Anomaly Detection

Say we have a record of how some value changes over time - hourly temperate in California, or number of visitors to a website per minute, for example. A lot of these kinds of records will have a regular shape or pattern that is indicative of some kind of "normal" behaviour. A deviation from this regular pattern - an *anomaly* - may indicate that something funny is going on. Maybe a load balancer has failed, or maybe a cluster user has submitted an overly-large job. If we can detect such anomalies automatically, then we can investigate as soon as the problem happens, rather than hours or days later when users start complaining.

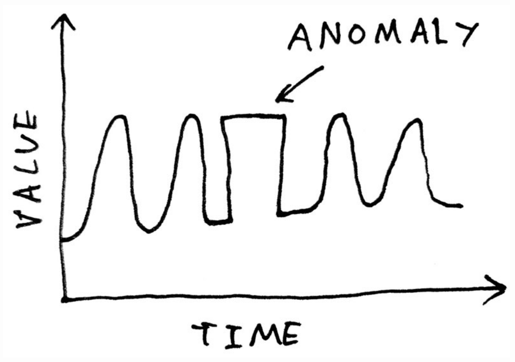
In some cases, we may be able to detect anomalies simply by looking for any values beyond a certain threshold:



In other cases, though, the structure of the waveform may prevent detection using this method:



And more subtle errors - a change in the shape of a periodic waveform, for examples - will be simply impossible to pick up with such a method:



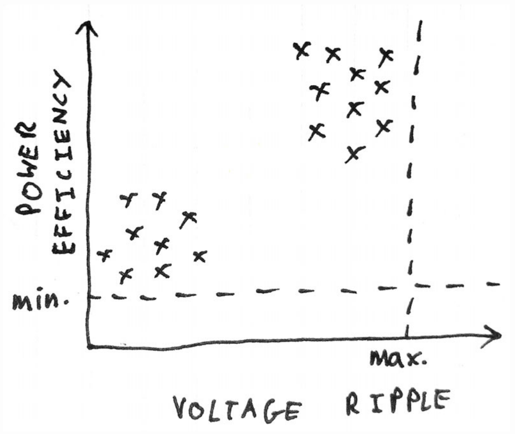
What we want is more a method that can learn what constitutes a "normal" waveform, not just in terms of its instantaneous value, but in terms of its *shape*.

Our approach will be to define an "anomaly" as being some pattern in the waveform that hasn't been seen before. Our algorithm will build up a library of "normal" waveform shapes, and use that library to try and reconstruct a waveform to be tested. If the reconstruction is poor, then the waveform is likely to contain something abnormal, and is therefore anomalous.

Clustering

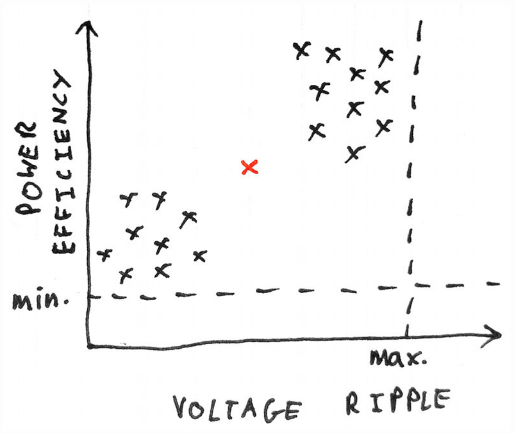
To explore how we might go about determining what counts as normal, let's take a step back from our problem and have a look at a simpler example.

Suppose we're manufacturing power supplies. Our power supplies use capacitors from two different manufacturers. During the QA phase we measure two variables for each unit tested: power efficiency and voltage ripple. We could plot a graph of one variable against another to get a sense of the distribution of the data:



These two variables have acceptance tolerances, as shown by the dotted lines. Within these limits, however, we find that the data tends to form two groups, or *clusters*, depending on which type of capacitor is used.

Suppose the next power supply that comes off the assembly line meets the tolerance requirements, but falls a way outside either of the two groups:



This is a sign that there might be something wrong with the unit; that it is in some way anomalous.

Here our clusters existed in the two-dimensional space defined by two variables (or two *features*, in machine learning parlance). But we can easily visualise how the approach can generalise to a three-dimensional space, defined by three features - or even an *n*-dimensional space, defined by *n* features. (The general case is a little harder to visualise, so just imagine a three-dimensional space and pretend there are more dimensions.)

Clusters can be identified programatically using a *clustering* algorithm. The particular method we'll be using is called *k-means clustering*. (See Cluster - K-means algorithm on Coursera if you're interested in more details of how the k-means algorithm works.)

Waveform Space

To apply such a technique to waveforms, we'll need to decide how we're going to define the space in which clusters will be formed.

First, we'll need to split the waveform into segments to give us separate "samples". But what will our *features* be - what are the variables that will define the *n*-dimensional space? We could go with standard measurements of each segment such as maximum, minimum and spread. But since our technique is generalisable to any number of dimensions, we can do something a bit more clever: we can take each element of the segment as a separate dimension. For a segment containing 32 time values, we define a 32-dimensional space. It is in this 32-dimensional space that we will form our clusters of waveform segments.

Waveform Anomaly Detection

In the example of the power supplies, we detected anomalous samples based on the distance from clusters. For our waveforms, we're going to take a slightly different approach so that we can visualise what's going on.

Consider our 32-dimensional waveform space. Each point in this space represents a possible waveform segment. Similar segments will cluster together. The middle of each cluster (the *centroid*) will provide some measure of the prototypical waveform pattern that all those segments are specific instances of. (If this is difficult to visualise, the other way to consider it is that the centroid is simply an average of all waveform samples in that cluster.)

Note that the centroid, being a point in the waveform space, is itself a waveform. Thus, the cluster centroids provide us with a set of "normal" waveform segments.

Suppose we then try to use our set of "normal" segments to reconstruct a set of data to be tested. If the data is similar in shape to what has come before it, we will be able to manage a good reconstruction. However, if the data contains some abnormal shape, we will not be able to reconstruct it using our normal shape library, and we will get a reconstruction error. This error will indicate an anomaly!

In summary, our algorithm will be:

Training:

- Split waveform data into segments of *n* samples
- Form a space in *n* dimensions, with each segment representing one point
- Determine clustering of segment points, and determine the centres or *centroids* of the clusters
- Cluster centroids provide library of "normal" waveform shapes

Testing:

- Try to reconstruct waveform data to be tested using cluster centroids learned during training
- Poor reconstruction error on any individual segment indicates anomalous shape

Our Data Set

To explore anomaly detection, we'll be using an EKG data set from PhysioNet, which is essentially the squishy version of the data we'll be getting from servers. Since this data has a very regular waveform, it provides a good vehicle for us to explore the algorithms without getting bogged down in the complications that come with real-world data.

The data is supplied in the `a02.dat` file. A Python module `ekg_data.py` is provided to read the data.

Exploring the Data

Let's get started by importing the EKG data module and examining what the data looks like:

In [1]:

```
from __future__ import print_function
import ekg_data

ekg_filename = 'a02.dat'
ekg_data = ekg_data.read_ekg_data(ekg_filename)
print(ekg_data.shape)
```

```
(3182000,)
```

The data is provided as a one-dimensional list of floating-point samples:

In [2]:

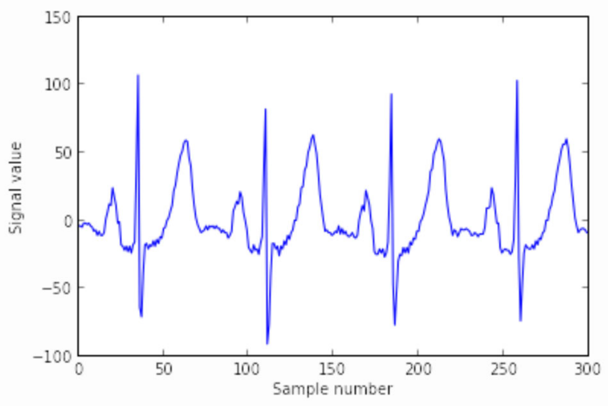
```
print("ekg_data[0]:\t", ekg_data[0])
print("ekg_data[1]:\t", ekg_data[1])
print("ekg_data.min:\t", ekg_data.min())
print("ekg_data.max:\t", ekg_data.max())
```

```
ekg_data[0]:     -4.0
ekg_data[1]:     -4.0
ekg_data.min:    -572.0
ekg_data.max:    580.0
```

We can also plot a number of samples from the data to get a feel for the waveform:

In [3]:

```
# IPython 'magic' command to set the matplotlib backend to display
# in the notebook
%matplotlib inline

import matplotlib.pyplot as plt

n_samples_to_plot = 300
plt.plot(ekg_data[0:n_samples_to_plot])
plt.xlabel("Sample number")
plt.ylabel("Signal value")
plt.show()
```



Since we have rather a lot of data, we'll take just the first 8,000 samples so our examples will run a bit faster:

In [4]:

```
ekg_data = ekg_data[0:8192]
```

Windowing

The first step in our process is to split the waveform into overlapping segments, with the section of the original data sampled sliding along by two samples each time. We take this approach so that we get instances of each waveform shape with a variety of horizontal translations.

In [5]:

```
import numpy as np

segment_len = 32
slide_len = 2

segments = []
for start_pos in range(0, len(ekg_data), slide_len):
    end_pos = start_pos + segment_len
    # make a copy so changes to 'segments' doesn't modify the original ekg_data
    segment = np.copy(ekg_data[start_pos:end_pos])
    # if we're at the end and we've got a truncated segment, drop it
    if len(segment) != segment_len:
        continue
    segments.append(segment)

print("Produced %d waveform segments" % len(segments))
```

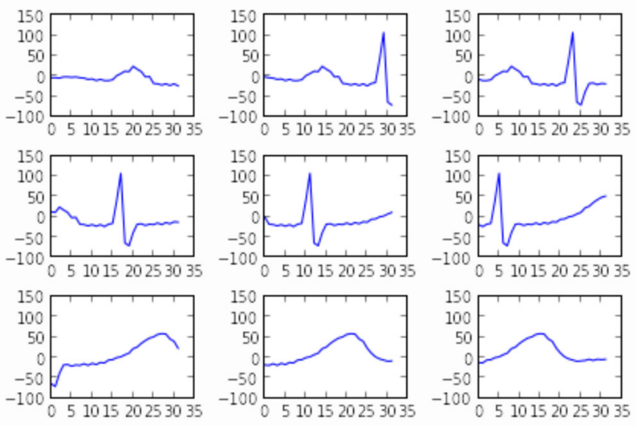
```
Produced 4081 waveform segments
```

(This code is saved for later reuse in `learn_utils.sliding_chunker`.)

Let's take a look at the segments we've produced:

In [6]:

```
import learn_utils

learn_utils.plot_waves(segments, step=3)
```

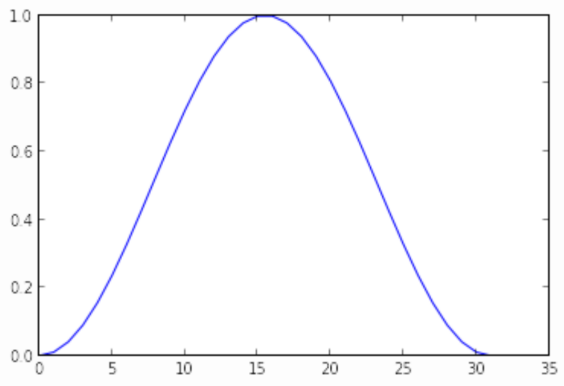


Note that these segments do not necessarily begin and end with a value of zero. This could be a problem later on: since the learned "normal" segment will then also have non-zero starts and end, when we try to reconstruct our waveform to be tested by adding together our learned segment, we'll end up with discontinuities.

The way we avoid this problem is to apply a *window function* to the data, which forces the start and end to be zero. A simple window function we can apply is the first half of a sine wave:

In [7]:

```
window_rads = np.linspace(0, np.pi, segment_len)
window = np.sin(window_rads)**2
plt.plot(window)
plt.show()
```


We then multiply each segment by this window function:

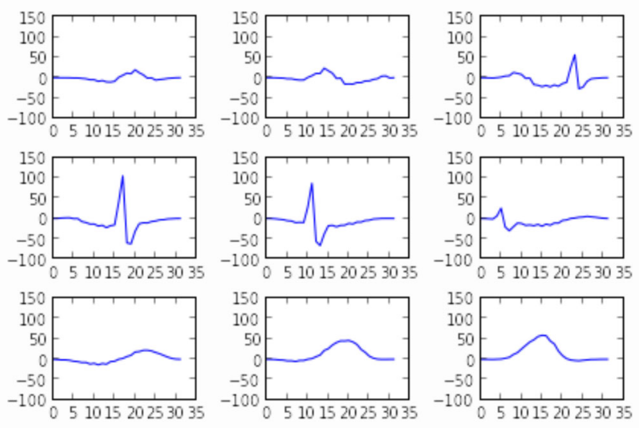
In [8]:

```
windowed_segments = []
for segment in segments:
    windowed_segment = np.copy(segment) * window
    windowed_segments.append(windowed_segment)
```

Plotting the result, we see the difference that the windowing process makes. The segments are now flat at the start and end - perfect to be joined together later.

In [9]:

```
learn_utils.plot_waves(windowed_segments, step=3)
```


Note that windowing also has the effect of making the segments less affected by the waveform either side of the segment. The waveform shape represented by the segment is now more "concentrated" in the middle.

Clustering

Next, we cluster our waveform segments in 32-dimensional space. The k-means algorithm is provided by Python's *scikit-learn* library.

In [10]:

```
from sklearn.cluster import KMeans

clusterer = KMeans(n_clusters=150)
clusterer.fit(windowed_segments)
```

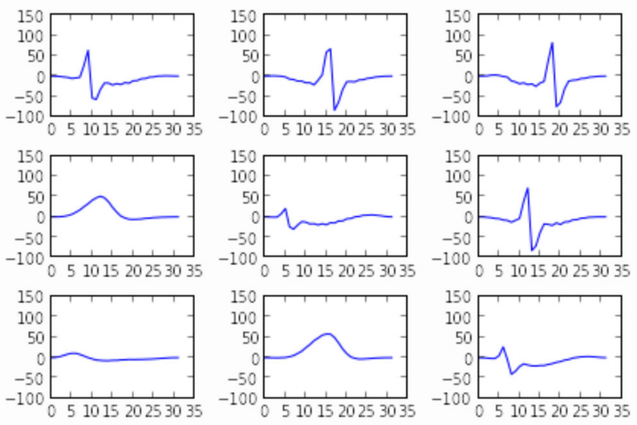
Out[10]:

```
KMeans(copy_x=True, init='k-means++', max_iter=300, n_clusters=150, n_init=10,
    n_jobs=1, precompute_distances='auto', random_state=None, tol=0.0001,
    verbose=0)
```

The cluster centroids are available through the `cluster_centers` attribute. Let's take a look to see what sort of shapes it's learned:

In [11]:

```
learn_utils.plot_waves(clusterer.cluster_centers_, step=15)
```



The clusterer appears to have learned a small number of basic shapes, with various horizontal translations.

Reconstruction from Clusters

Finally, we come to the interesting part of the algorithm: reconstructing our waveform to be tested using the learned library of shapes. Our approach is going to be very simple. We'll:

- Split the data into overlapping segments
- Find the cluster centroid which best matches our segment
- Use that centroid as the reconstruction for that segment
- Join the reconstruction segments up to form the reconstruction

This time, we only need enough overlap between segments to allow us to stitch the reconstructions back together again, so we'll go with an overlap of half a segment.

First, let's see how well we do at reconstructing the original waveform. We first form segments:

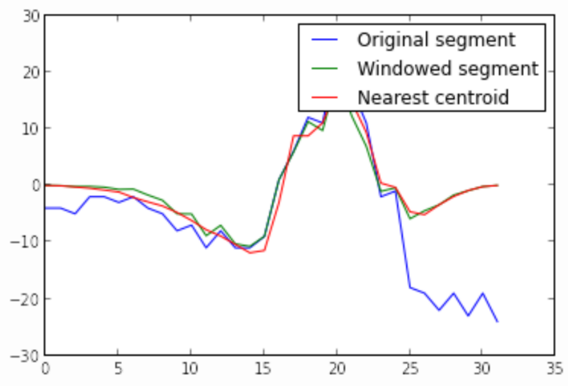
In [12]:

```
slide_len = segment_len/2
test_segments = learn_utils.sliding_chunker(
    ekg_data,
    window_len=segment_len,
    slide_len=slide_len
)
```

Before trying to reconstruct the whole thing, we can check how the reconstruction fares for individual segments. Try changing the segment index (if you're running an interactive version of the notebook) to see the reconstruction for different segments.

In [13]:

```
centroids = clusterer.cluster_centers_

segment = np.copy(test_segments[0])
# remember, the clustering was set up using the windowed data
# so to find a match, we should also window our search key
windowed_segment = segment * window
# predict() returns a list of centres to cope with the possibility of multiple
# samples being passed
nearest_centroid_idx = clusterer.predict(windowed_segment)[0]
nearest_centroid = np.copy(centroids[nearest_centroid_idx])
plt.figure()
plt.plot(segment, label="Original segment")
plt.plot(windowed_segment, label="Windowed segment")
plt.plot(nearest_centroid, label="Nearest centroid")
plt.legend()
plt.show()
```

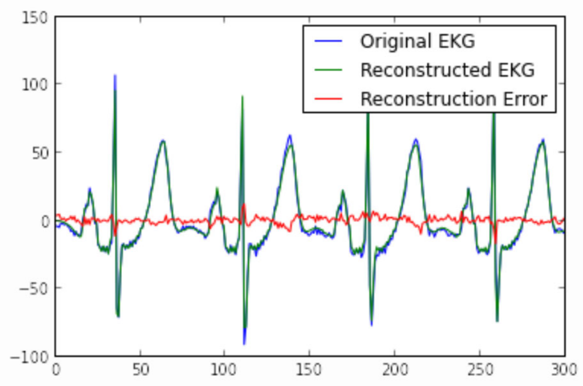


Looking good! So now, let's go ahead and try and reconstruct the full set of data.

In [14]:

```
reconstruction = np.zeros(len(ekg_data))
slide_len = segment_len/2

for segment_n, segment in enumerate(test_segments):
    # don't modify the data in segments
    segment = np.copy(segment)
    segment *= window
    nearest_centroid_idx = clusterer.predict(segment)[0]
    centroids = clusterer.cluster_centers_
    nearest_centroid = np.copy(centroids[nearest_centroid_idx])

    # overlay our reconstructed segments with an overlap of half a segment
    pos = segment_n * slide_len
    reconstruction[pos:pos+segment_len] += nearest_centroid

n_plot_samples = 300

error = reconstruction[0:n_plot_samples] - ekg_data[0:n_plot_samples]
error_98th_percentile = np.percentile(error, 98)
print("Maximum reconstruction error was %.1f" % error.max())
print("98th percentile of reconstruction error was %.1f" % error_98th_percentile)

plt.plot(ekg_data[0:n_plot_samples], label="Original EKG")
plt.plot(reconstruction[0:n_plot_samples], label="Reconstructed EKG")
plt.plot(error[0:n_plot_samples], label="Reconstruction Error")
plt.legend()
plt.show()
```

```
Maximum reconstruction error was 12.4
98th percentile of reconstruction error was 5.6
```



(This reconstruction code is stored in `learn_utils.reconstruct` for later reuse.)

The reconstruction is not bad at all!

Anomaly Detection

Now that we're confident in our reconstruction algorithm, let's see what happens when we have an anomalous waveform. We'll introduce the anomaly manually by zeroing out a small number of samples of the original:

In [15]:

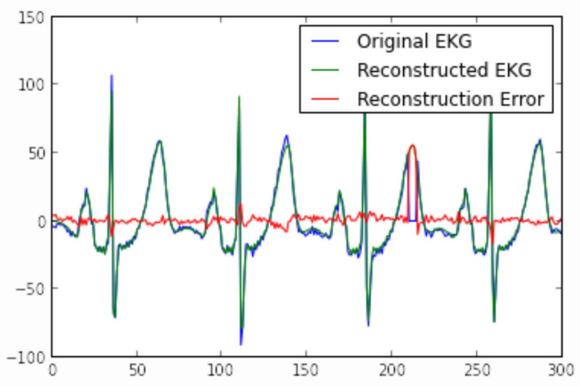
```
ekg_data_anomalous = np.copy(ekg_data)
ekg_data_anomalous[210:215] = 0
```

Then attempting a reconstruction of this data, using the same code as above:

In [16]:

```
recontruction = \
    learn_utils.reconstruct(ekg_data_anomalous, window, clusterer)

error = reconstruction[0:n_plot_samples] - ekg_data_anomalous[0:n_plot_samples]
error_98th_percentile = np.percentile(error, 98)
print("Maximum reconstruction error was %.1f" % error.max())
print("98th percentile of reconstruction error was %.1f" % error_98th_percentile)

plt.plot(ekg_data_anomalous[0:n_plot_samples], label="Original EKG")
plt.plot(reconstruction[0:n_plot_samples], label="Reconstructed EKG")
plt.plot(error[0:n_plot_samples], label="Reconstruction Error")
plt.legend()
plt.show()
```

```
Maximum reconstruction error was 55.6
98th percentile of reconstruction error was 9.5
```



Since our anomaly has produced a shape in the waveform that hadn't been seen before, the waveform around that point couldn't be reconstructed using the learned shape library. This gives a large, easily visible reconstruction error! This error could be easily detected using a simple threshold detector.

Conclusion

We've looked at one example of the use of unsupervised learning techniques in systems administration: anomaly detection of time series data based on reconstruction error from k-means clustering.

This problem could have been solved in different ways. For example, we could have trained a predictive neural network model, and examined the difference between the predicted waveform and the actual waveform. In general, there is no 'one size fits all' solution in machine learning - different techniques will be required for different problems.

A standalone script containing our full set of code is available at https://github.com/mrahtz/sanger-machine-learning-workshop as `learn.py`.

22 Comments Amid Fish Disqus' Privacy Policy Login

Favorite 3 Tweet Share Sort by Best

Join the discussion…

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name

Jayanti Prasad • 2 years ago
This is a really a very important, useful and interesting tutorial I have come across about outliers detection with K-Means. Thanks for putting the efforts.
Reply • Share ›

Heldie :) • 3 years ago • edited
How to resolve " ModuleNotFoundError: No module named 'ekg_data' "
and also " No module named 'learn_utils "
pls can anyone help me with this .... thanks in adv :)
Reply • Share ›

苗嘉澍 → Heldie :) • 3 years ago
in your command line/ terminal

pip install ekg_data
pip install learn_utils

or in your jupyter notebook where you write your python code

!pip install ekg_data
!pip install learn_utils
Reply • Share ›

Milan Harkhani • 4 years ago
In conclusion, you've mentioned about other technique that we could have trained a predictive neural network model, and examined the difference between the predicted waveform and the actual waveform. so how its works? for example I have machine's sensors data and goal is to predict machine fault. what is the approach here. should I train my neural net with normal data (data without anomalies) and predict each sensor's behaviour?. is the neural net predict normal behaviour of particular sensor when anomaly occur ? and we calculate residual using predicted and actual data of the sensor. I'm very confused. can you explain what is the actual technique?
Reply • Share ›

Mc Kayne → Milan Harkhani • 4 years ago
Beside clutering anomaly detection I work also on this area. First you need to "lag" your time series which means you create a table with time shifts. So your table contains the value on time t and (with lag 3) the three values from the time steps before (t-1, t-2, t-3). This schema is done for every time point in your time series. With this table you can enter a neural network or every other regression predictor. The neural network adapts the weights by learning which must be the target value by given values in the past. Now the network is ready for a prediction. It is important here, that the network absolutely needs an input on which the prediction can be done. So if you have some values in the past, you can predict (based on the learned model) the next (unknown) value.
Reply • Share ›

Milan Harkhani → Mc Kayne • 4 years ago
thanks Mc Kayne, I understand the method you suggest. my question is what output will ANN gives when anomalous data is given? A normal estimated output(output value which is normal as given in training data). For example in RNN(replicator neural net) gives pattern reconstruction error. what I want as output is system normal behaviour when anomalous data is given. so later I can calculate residual and set threshold value.
Reply • Share ›

Mc Kayne → Milan Harkhani • 4 years ago
Yes you are right. That's also my aim to built up an NN which gives as output the normal behaviour thus I can calculate the difference to an abnormal signal. But the biggest problem to solve here is that the NN inevitable needs an input to do

the biggest problem to solve here is that the NN inevitable needs an input to do a prediction/forecast for the next point. So you can't just let the NN forecast you the next thousand points. For each point in the future you need also an input. Question here is, how far into the future you can look with an NN? If you find an solution let me know :)

^ | v 1 • Reply • Share ›

Milan Harkhani → Mc Kayne • 4 years ago

hey Mc Kayne, thanks for help. I've tried lagged NN but not getting right result. could you share some related stuffs like blog, example or book anything to this particular problem ?

^ | v • Reply • Share ›

Mc Kayne → Milan Harkhani • 4 years ago

Hi Milan,

here is a link to the KNIME Forum: https://forum.knime.com/t/t...

The person here tried to face the same problem I mentioned above. In my person I didn't worked any more on this problem, because for me a simple autoregression with a NN is good enough. In the Internet you find tons of stuff reffering to times series prediction/forecasting. If the regression with a "simple" feedforward NN isn't accurate enough try out a RNN or LSTM Network (which I am doing now)

^ | v • Reply • Share ›

Mc Kayne • 4 years ago

One question: I used this anomaly detection algorithm to detect anomalies in a real life manufacturing process (times series of torque from a production machine). Altough the process is always the same, the time series has some delays. This means, the window sliding over the series sees always another process detail and thus can't capture the pattern of it. How would you face this problem of inconsistent time series with the clustering approach?

^ | v • Reply • Share ›

Matthew Rahtz Mod → Mc Kayne • 4 years ago

Could you clarify what you mean by "always sees another process detail"?

^ | v • Reply • Share ›

Mc Kayne → Matthew Rahtz • 4 years ago

View — uploads.disquscdn.com

With "always sees another process detail" I mean the problem illustrated in the picture. If I don't work with 150 cluster centroids but only with 4 centroids (should be enough to capture the basic waveforms) I ran in the mentioned problem. Maybe you don't get this issue because you work with so many cluster centroids. Hope you understand my problem about the sliding window and the inconsistent time series.
Greets

^ | v • Reply • Share ›

Matthew Rahtz Mod → Mc Kayne • 4 years ago

I see what you mean. One option might be to try and set the window position manually - for example, you could use a peak detector, and then place the windows so that peaks were always in the middle of the window. There might be some overlap between windows, but that should be fine if the waveform is really as regular as your demonstration.

^ | v • Reply • Share ›

Mc Kayne → Matthew Rahtz • 4 years ago

Yeah I understand your approach. It's like doing window sliding with adjustable window size/position. My time series looks like in the picture below. Do you mean the peak detector would be a solution for my problem here? Btw do you know an example code for python? Searched a bit bit didn't found the right thing. Thanks and greets.

View — uploads.disquscdn.com

^ | v • Reply • Share ›

Matthew Rahtz Mod → Mc Kayne • 4 years ago

Yeah, I think you might be able to use a peak detector to adjust the window position. I don't know of any example code, but it shouldn't be too hard to hack one up yourself - play around and see what works.

one up yourself - play around and see what works.

Reply • Share ›

m b • 4 years ago

How would you displays the clusters (i mean the centroids with cluster's content) ? Since you have a 32D space i don't see any easy solution.

Reply • Share ›

Matthew Rahtz Mod m b • 4 years ago

The usual approach is to apply a 'dimensionality reduction' algorithm to reduce the dimensionality to 2D, then plot that. http://colah.github.io/post... has a really nice explanation.

1 • Reply • Share ›

m b Matthew Rahtz • 4 years ago • edited

thank you very much. Didn't knew this, it gave me quite nice results.

Reply • Share ›

Majid al-Dosari • 7 years ago

hmmm. can reconcile what you did after you read "A Review of Subsequence Time Series Clustering" which is meant to understand the paper that started it all "“Clustering of streaming time series is meaningless,”

Reply • Share ›

Matthew Rahtz Mod Majid al-Dosari • 7 years ago

Sorry for the slow reply - it's been a busy term. Eamonn did actually get in touch with me about his and Lin's findings in "Clustering of Time Series Subsequences is Meaningless" and we talked about it a little bit. The conclusion I came to is that maybe it works here because this is a slightly different case than classical clustering. The aim here isn't necessarily to find a restricted number of good motifs, but to find a set of basis waveforms characteristic to non-anomalous data. From what I understood, the biggest problem pointed out in Eamonn's work is that in subsequence clustering there must exist a weighted sum of the cluster centres that results in a straight line, and since there's no reason to expect the desired motifs of a signal to obey this property, clustering will not find those motifs - but maybe that doesn't matter here, since discovery of motifs is not the aim. Eamonn suggested that another factor may be the abnormally large number of cluster centres used in this approach.

I had a quick skim of the paper you mentioned, although I haven't looked at the specifics of the problems in the pre-proof algorithms. Was there a particular aspect of the review that you thought might be relevant in explaining or refuting the behaviour seen here?

1 • Reply • Share ›

Majid al-Dosari Matthew Rahtz • 7 years ago

i don't see a big difference between motif discovery and finding basis waveforms (?).

if you ask me i would not use any clustering technique on time series data for the task of anomaly detection. there are just too many unanswered questions and parameters to worry about. 150 clusters for simple looking data? then you are just using one centroid/basis to reconstruct? are you sure you didn't just memorize the data? maybe b/c you're memorizing the data you don't have any issue? if you are using an unsupervised setting then you might have an anomaly in your data and it would get memorized if you're using so many clusters. how do you know there is no anomaly in the data when training? i realize you're using this material to educate but it just leaves too many questions.

but i can understand the use of clustering to try to organize and understand the data..perhaps w/o the use of sliding windows to mitigate the problems presented in that paper.

if you want to capture some notion of basis waveforms to reconstruct from, check out the stuff from signals processing like https://en.wikipedia.org/wi.... this is especially applicable for the type of data in your example. i suspect that these techniques are more robust to noise as well as having anomalies in them.

Reply • Share ›

Matthew Rahtz Mod Majid al-Dosari • 7 years ago

What I was thinking is that, from what I understand, with motif discovery you're looking for a sparse code to describe the data, so you want to keep the number of clusters to a minimum, whereas here because sparsity isn't the objective, you can let the number of clusters be much larger - and because of the larger number, maybe the set of cluster centres can include waveforms such that the condition from Eamonn's paper is satisfied but the waveforms are still useful for anomaly detection with this reconstruction error method. I'm not sure how rigorous this intuition is, though.

But you're right, there are a number of unanswered questions about why this is working and how well it's going to work in practice, so I've added a warning at the top of the post about this.

Reply • Share ›

Amid Fish

is Matthew Rahtz's blog

GitHub, LinkedIn, or say hello at matthew.rahtz@gmail.com!