

# Tutorial 5 — catch22 feature extraction

“ Auto-generated from [tutorial\\_notebook/5\\_Ethoscopy\\_catch22\\_tutorial.ipynb](#).  
Executed against the seaborn canvas so every figure is inline as a static PNG. Plotly-only cells are kept for context and marked as placeholders — for the interactive version, run the source notebook.

## Ethoscopy - Behavpy to catch22

Catch22 is a shortened version of HCTSA, a time series comparative analysis toolbox in matlab. Catch22 has been adapted to also work in python and uses the top 22 most used analytical tests from the 100s used in HCTSA.

See heere for more details: <https://feature-based-time-series-analys.gitbook.io/catch22-features/>

or their github:

<https://github.com/DynamicsAndNeuralSystems/pycatch22>

### 1. Load the dummy dataset

```
import ethoscopy as etho
import pandas as pd
import numpy as np
import pycatch22
```

# One-time setup: fetch the tutorial datasets

The tutorial pickle files (~36 MB total, dominated by `overview_data.pkl`) are **not** bundled with the PyPI package, to keep `pip install ethoscopy` lean.

Run the cell below **once per environment** to download them into the installed package directory. Subsequent runs are idempotent (already-present files are skipped).

You can also fetch them manually from

[https://github.com/gilestrolab/ethoscopy/tree/main/src/ethoscopy/misc/tutorial\\_data](https://github.com/gilestrolab/ethoscopy/tree/main/src/ethoscopy/misc/tutorial_data).

```
# Idempotent: skips files that are already present.
import ethoscopy as etho
etho.download_tutorial_data()
```

```
[skip] overview_data.pkl (already present)
[skip] overview_meta.pkl (already present)
[skip] circadian_data.pkl (already present)
[skip] circadian_meta.pkl (already present)
[skip] 4_states_F_WT.pkl (already present)
[skip] 4_states_M_WT.pkl (already present)
Tutorial data ready in: /home/gg/.cache/ethoscopy/tutorial_data
```

```
PosixPath('/home/gg/.cache/ethoscopy/tutorial_data')
```

```
# Load in the data from the circadian tutorial
from ethoscopy.misc.get_tutorials import get_tutorial
data, metadata = get_tutorial('circadian')
df = etho.behavpy(data, metadata, check = True)
```

## 2. Some data curation

```
# We'll look at movement here
var = 'moving'
```

```
# The most basic curation is to pick a specific time period
df = df.t_filter(start_time = 0, end_time = 9*24)
```

```
# We can also use interpolate to fill in the missing data points, this can be useful if it's
only a few points missing per specimen
df = df.interpolate(variable = var, step_size = 60, t_column = 't')
```

```
# Or we can also group several rows together by increasing the t diff, here we increase
# from 60 to 120, so we find the average of every two rows
```

```
## We won't run this here, but keep it in mind for the future if you have too few data points
# df = df.bin_time(column = var, bin_secs = 120, function = 'mean')
```

```
# Call .summary() to check the amount of data points
df.summary(True)
```

	data_points	time_range
id		
2017-01-16 08:00:00 circadian.txt 01	12000 57600.0	-> 777540.0
2017-01-16 08:00:00 circadian.txt 02	12000 57600.0	-> 777540.0
2017-01-16 08:00:00 circadian.txt 03	12000 57600.0	-> 777540.0
2017-01-16 08:00:00 circadian.txt 04	12000 57600.0	-> 777540.0
2017-01-16 08:00:00 circadian.txt 05	12000 57600.0	-> 777540.0
2017-01-16 08:00:00 circadian.txt 06	12000 57600.0	-> 777540.0
2017-01-16 08:00:00 circadian.txt 07	12000 57600.0	-> 777540.0
2017-01-16 08:00:00 circadian.txt 08	12000 57600.0	-> 777540.0
2017-01-16 08:00:00 circadian.txt 09	12000 57600.0	-> 777540.0
2017-01-16 08:00:00 circadian.txt 10	12000 57600.0	-> 777540.0
2017-01-16 08:00:00 circadian.txt 11	12000 57600.0	-> 777540.0
2017-01-16 08:00:00 circadian.txt 12	12000 57600.0	-> 777540.0
2017-01-16 08:00:00 circadian.txt 13	12000 57600.0	-> 777540.0
2017-01-16 08:00:00 circadian.txt 14	12000 57600.0	-> 777540.0
2017-01-16 08:00:00 circadian.txt 15	12000 57600.0	-> 777540.0

2017-01-16 08:00:00 circadian.txt 16	12000	57600.0	->	777540.0
2017-01-16 08:00:00 circadian.txt 17	12000	57600.0	->	777540.0
2017-01-16 08:00:00 circadian.txt 18	12000	57600.0	->	777540.0
2017-01-16 08:00:00 circadian.txt 19	12000	57600.0	->	777540.0
2017-01-16 08:00:00 circadian.txt 20	12000	57600.0	->	777540.0
2017-01-16 08:00:00 circadian.txt 21	12000	57600.0	->	777540.0
2017-01-16 08:00:00 circadian.txt 22	12000	57600.0	->	777540.0
2017-01-16 08:00:00 circadian.txt 23	12000	57600.0	->	777540.0
2017-01-16 08:00:00 circadian.txt 24	12000	57600.0	->	777540.0
2017-01-16 08:00:00 circadian.txt 25	12000	57600.0	->	777540.0
2017-01-16 08:00:00 circadian.txt 26	12000	57600.0	->	777540.0
2017-01-16 08:00:00 circadian.txt 27	12000	57600.0	->	777540.0
2017-01-16 08:00:00 circadian.txt 28	12000	57600.0	->	777540.0
2017-01-16 08:00:00 circadian.txt 29	12000	57600.0	->	777540.0
2017-01-16 08:00:00 circadian.txt 30	12000	57600.0	->	777540.0
2017-01-16 08:00:00 circadian.txt 31	12000	57600.0	->	777540.0
2017-01-16 08:00:00 circadian.txt 32	12000	57600.0	->	777540.0

```
# Once we've completed our curation or if we just want to remove specimens that don't have
enough values,
# we can call curate to remove all specimens with too few points still
# We can see from summary() that the normal amount is 11999
df = df.curate(points = 11999)

# Note: The interpolate method returns rows 1 shorter than before so you'll need to add a
minus 1 if using curate after
# Note: If you've called the above bin_time this curate will return an empty dataframe
```

```
# When using x position data the interesting part is how the fly positions itself in relation
to the food
# However this will be different on the x,y axis for flies on either side of the ethoscope, so
lets normalise it
# You only need to run this is using the x variable, which we aren't right now
# df_r = df.xmv('region_id', list(range(11,21)))
# df_l = df.xmv('region_id', list(range(1,11)))
# df_r['x'] = 1 - df_r['x']
# df = df_l.concat(df_r)
```

# 3. Normalise the data

The ethoscope data can sometimes do with some augmentation to make sure it performs better in Catch22

Try running your data unnormalised and normalised, and with different methods to see the results

```
# Catch22 takes only lists, but to normalise the data we'll have to put it into a numpy array first
list_x = df.groupby(df.index, sort = False)[var].apply(list)
arr_x = np.array([np.array(x) for x in list_x])
# Here we grab the ids of each for the labels that we'll use later
list_id = list_x.index.tolist()
```

```
# Use some or all of these functions to normalise the data between specimens

# norm01 transforms the data to be between 0 and 1
def norm01(x):
    return (x-np.nanmin(x))/(np.nanmax(x)-np.nanmin(x))
# or
# find the zscore
def zscore(x):
    return (x-np.mean(x))/(np.std(x))

# Only use this if looking at phi, it changes it be only from 0-90 or horizontal to vertical
as the ethoscope doesn't track direction
def norm_phi(x):
    return np.where(x > 90, 90 - (x - 90), x)

# Smooth out the time series data
def moving_average(a, n) :
    ret = np.cumsum(a, dtype=float)
    ret[n:] = ret[n:] - ret[:-n]
    return ret[n - 1:] / n
```

As the data here is just 1s and 0s (True and False) we don't want to perform any curation. Head to tutorial 6 for information on how to apply it to other sets of data. But for now just bear in mind that it's a useful pre-processing tool.

## 4. Run Catch22 and store in a behavpy dataframe

```
# Here the time series data is augmented to fit into the correct format, a nested list, and
run through the catch22 function: pycatch22.catch22_all()
# You can also call individual tests if you know which ones you need, see the pycatch
documents for more details
# If you want the mean and std then call catch24 = True within the function
data = [pycatch22.catch22_all(list_x[i])['values'] for i in range(len(list_x))]
cols = pycatch22.catch22_all(list_x[0])['names']

c22 = pd.DataFrame(data, columns = cols, index = list_id)
```

```
/tmp/ipykernel_557100/842141969.py:4: FutureWarning: Series.__getitem__ treating keys as
positions is deprecated. In a future version, integer keys will always be treated as labels
(consistent with DataFrame behavior). To access a value by position, use `ser.iloc[pos]`
data = [pycatch22.catch22_all(list_x[i])['values'] for i in range(len(list_x))]
```

```
/tmp/ipykernel_557100/842141969.py:5: FutureWarning: Series.__getitem__ treating keys as
positions is deprecated. In a future version, integer keys will always be treated as labels
(consistent with DataFrame behavior). To access a value by position, use `ser.iloc[pos]`
cols = pycatch22.catch22_all(list_x[0])['names']
```

To compare the scores and find trends we need to normalize and standardise the data

```
# sklearn is a common python package for statistics and machine learning, if you dont have in
installed, install from pip now.
```

```
from sklearn.preprocessing import normalize, MinMaxScaler
```

```
# We call the normalisation across each variable (column)  
# normalize() scales vectors to a unit norm so that the vector has a length of 1  
normed = normalize(c22, axis=0)
```

```
# We scale the data so all scores are fitted to between 0 and 1 (classification techniques  
perform much better with same scaled data)  
scaled = MinMaxScaler().fit_transform(normed)
```

```
# Lets put the data into a dataframe and then a behavpy dataframe  
c22_norm = pd.DataFrame(scaled, columns = c22.columns, index = list_id)  
c22_norm.index.name = 'id'  
c22_df = etho.behavpy(c22_norm, df.meta)
```

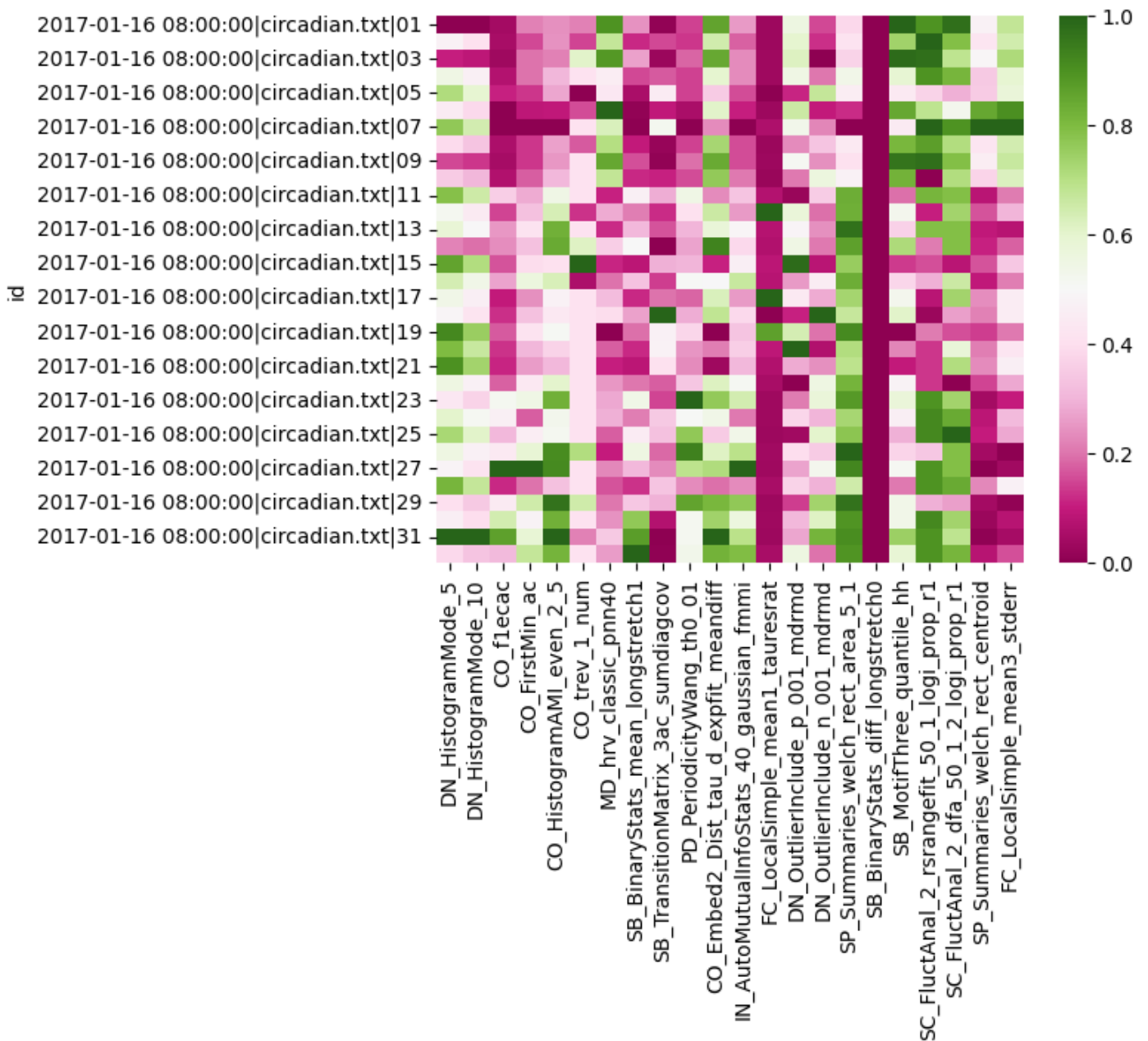
## 5. Viewing the results

```
# We'll use matplotlib and its add on pacakage seaborn here to create graphs rather than the  
usual plotly  
from matplotlib import pyplot as plt  
# You may need to pip install seaborn  
import seaborn as sns
```

Heatmaps are often the best way to get a quick view of where your outputs vary

```
heatmap = sns.heatmap(c22_df, cmap='PiYG')  
heatmap.set_title('Heatmap of normed output', fontdict={'fontsize':12}, pad=12);
```

Heatmap of normed output



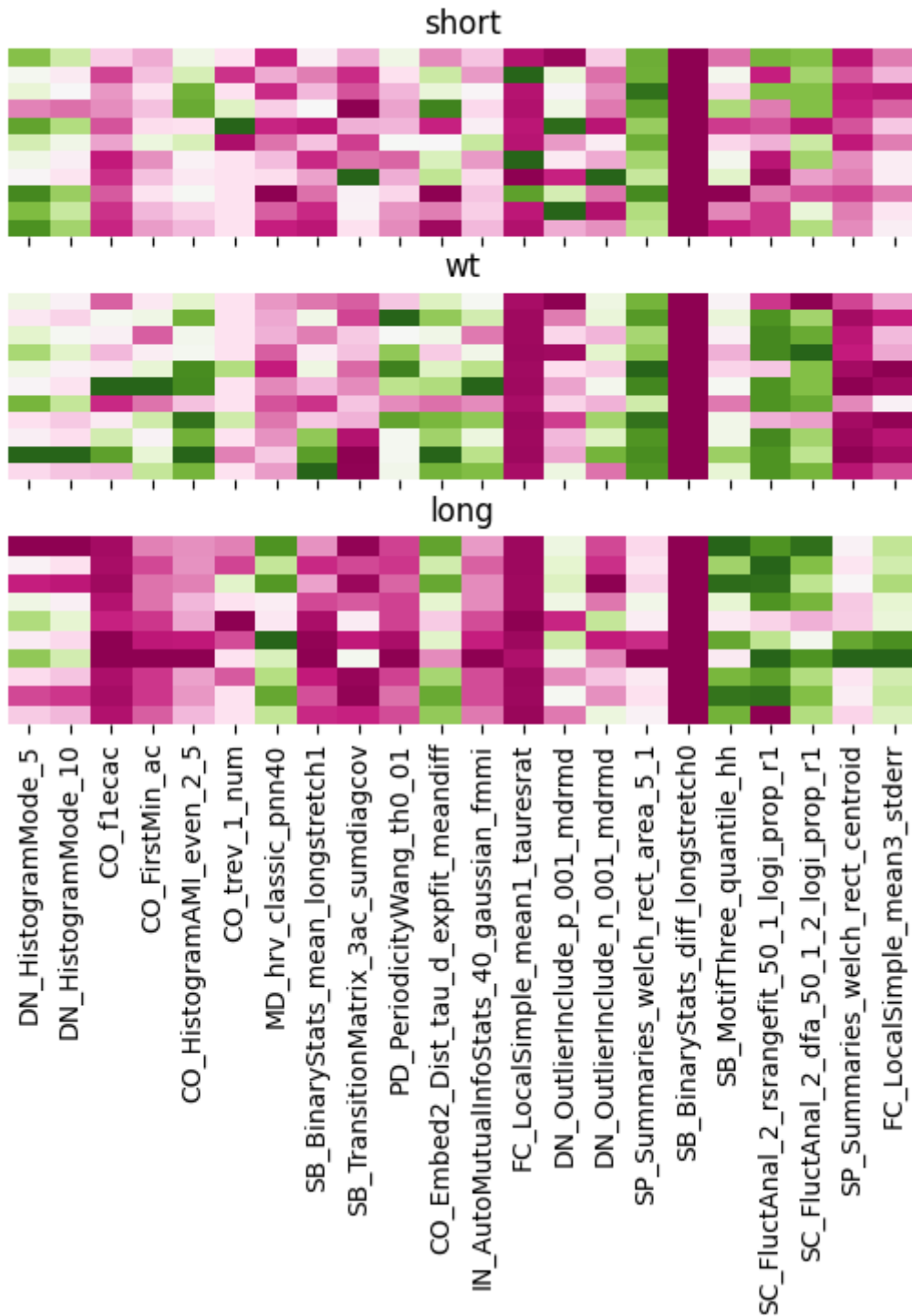
We can see a the two histogram mode tests all score roughly the same, so they're not likely to be useful. However, both the CO seem to have a variation in scores across the board. Lets look at the heatmap for each group we have in the metadata for circadian length: short, WT, and long

```
# We iterate and filter with xmv to get each group, removing the y ticks so its less cluttered

f, ax = plt.subplots(3,1, sharex='col')
```

```
f.subplots_adjust(hspace=0.3)
```

```
for c, typ in enumerate(['short', 'wt', 'long']):  
    tmp_df = c22_df.xmv('period_group', typ)  
    g = sns.heatmap(tmp_df, cmap="PiYG", cbar=False, ax=ax[c])  
    g.set(yticklabels=[])  
    g.set(title=typ)  
    g.set(ylabel=None)  
    g.tick_params(left=False)
```

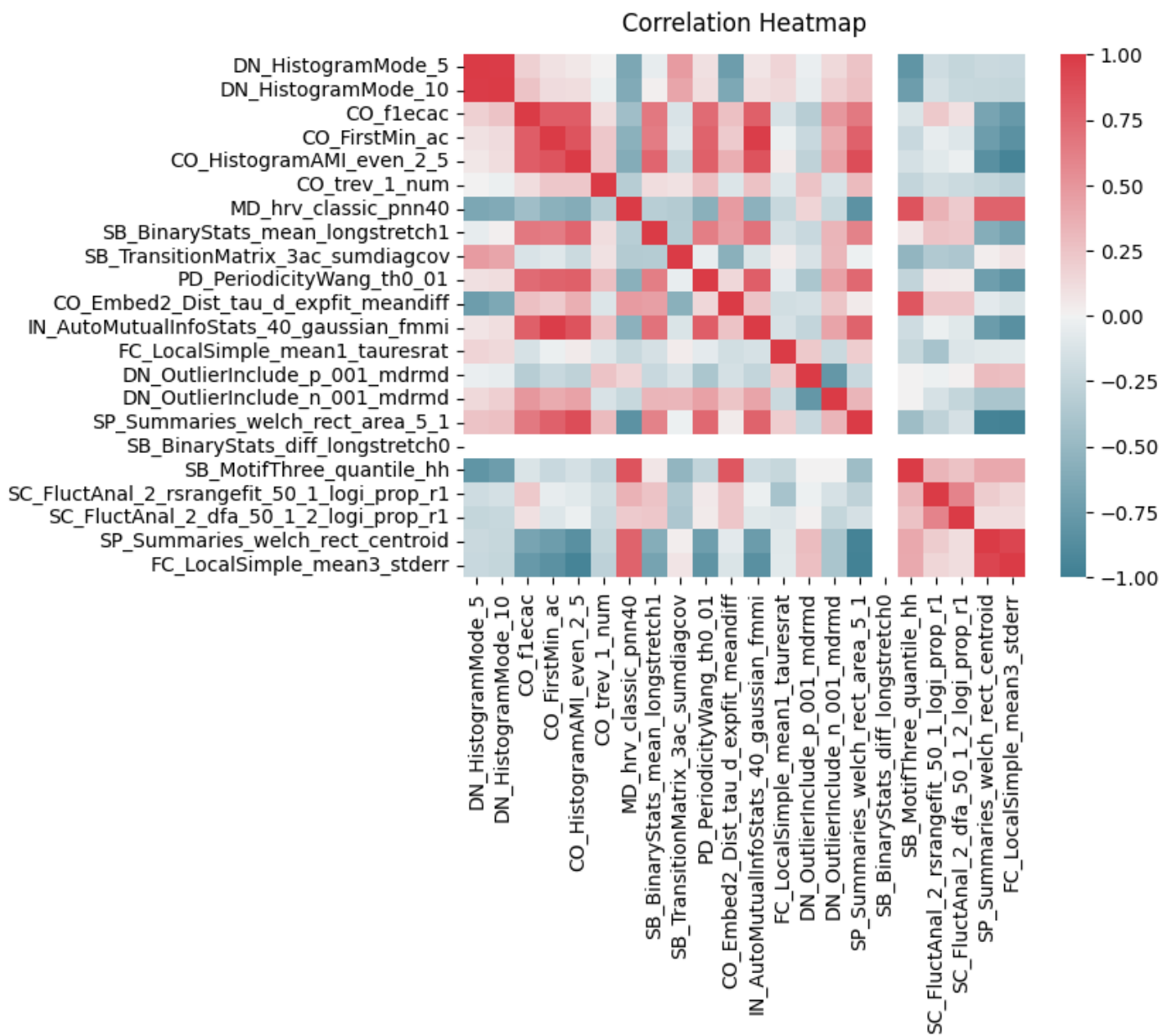


We can see clearly the difference between the groups with the CO analysis. As well as big differences between long to WT and short when looking at FC local Simple.

```
# We can also plot the correlation bewteen each variable to see the relationship between them
all

heatmap = sns.heatmap(c22_df.corr(), vmin=-1, vmax=1, cmap = sns.diverging_palette(220, 10,
as_cmap=True))

heatmap.set_title('Correlation Heatmap', fontdict={'fontsize':12}, pad=12);
```



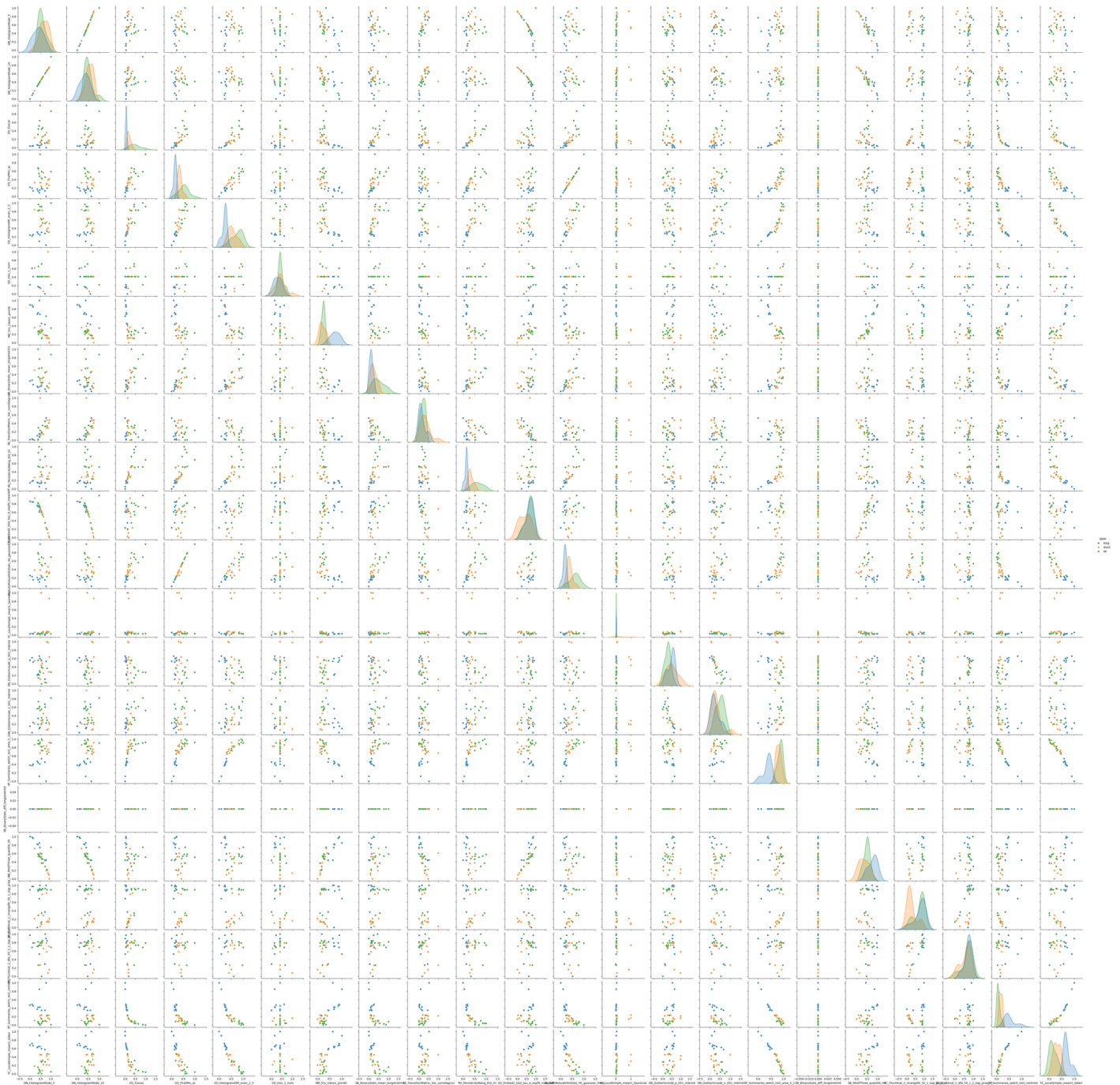
There are strong clusters of tests that are similar to each other which is to be expected. But we can also see CO tests are strong correlated both positively and negatively with many of the others

```
# We'll need the labels to colour code the plot, but to make sure they have the same order
we'll match them to the main df and remove them
# The order in the metadata is often (and should be) the same as order in the data, but
sometimes with curation it can change so this is just to be safe
# We make a dictionary of the id and the label and map it to data
label_dict = c22_df.meta['period_group'].to_dict()
c22_df['label'] = c22_df.index.to_series().map(label_dict)
```

A pairwise plot is a more visual version of the correlation plot, but we can also colour by our labels and see how they group

```
# Pairwise plot will give you the most insight into which variables have distinct populations
per label
sns.pairplot(c22_df, vars = cols, hue = 'label')
```

```
<seaborn.axisgrid.PairGrid at 0x7fbcfb652ba0>
```



We can see just visually from the pairwise plot what most of the variable combinations separate the populations into roughly distinct groups, which bodes well for classification. We can even see a few with linear correlations such as IN\_AutoMutualInfoStats\_40\_gaussian\_fmfi X CO\_FirstMin\_ac.

From the pairwise plot we can see that the tests can likely separate the groups well. We can visualise this all together with a PCA or TSNE plots. Both these plots take all the variables and reduce them to a lower dimensionality at the cost of some detail, often 2 or 3 as we can then visualise them. In the reduction the algorithm will create new components that represent the most valuable variables.

There are a few differences between PCA and TSNE, but in short TSNE works better with non-linear relationships and can group nearby points better. However, it does this at the cost of true distance between groups and over-represents differences between clusters.

```
# import both functions from sklearn
from sklearn.manifold import TSNE
from sklearn.decomposition import PCA
```

```
# lets turn each row into a list and nest them, we also need to drop the labels as they are
not numerical
X = c22_df.drop(columns = ['label']).values.tolist()
X = np.array([np.array(i) for i in X])
```

```
# We'll initialise the PCA function and state we want it reduced to 2 variables
pca = PCA(n_components=2)
# fit_transform() learns the data and then transforms it to the 2 principle components.
X_pca = pca.fit_transform(X)
```

```
# Turn the output into a pandas dataframe and add the labels
pca_df = pd.DataFrame(X_pca, columns = ['PCA1', 'PCA2'])
pca_df['label'] = c22_df['label'].tolist()
```

```
plt.figure(figsize=(10,4))

sns.scatterplot(data=pca_df,
```

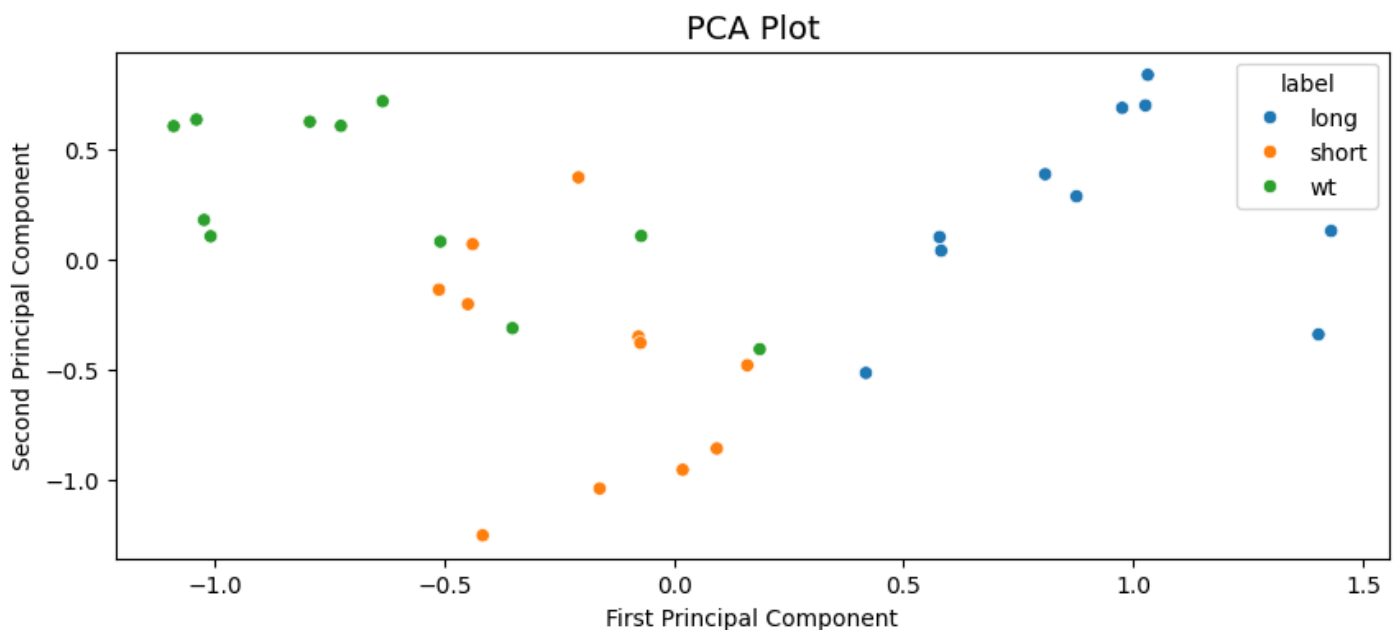
```

x="PCA1",
y="PCA2",
hue="label")

plt.title("PCA Plot",
          fontsize=14)
plt.xlabel('First Principal Component',
           fontsize=10)
plt.ylabel('Second Principal Component',
           fontsize=10)

```

```
Text(0, 0.5, 'Second Principal Component')
```



We can see the PCA analysis has 3 distinct groups with some of the WT labelled flies in the short cluster. All long labelled flies are separate and easy to distinguish.

```

# Repeat what we did above but with the TSNE function
tsne = TSNE(n_components=2, random_state=42)
X_ts = tsne.fit_transform(X)
ts_df = pd.DataFrame(X_ts, columns = ['TS1', 'TS2'])
ts_df['label'] = c22_df['label'].tolist()

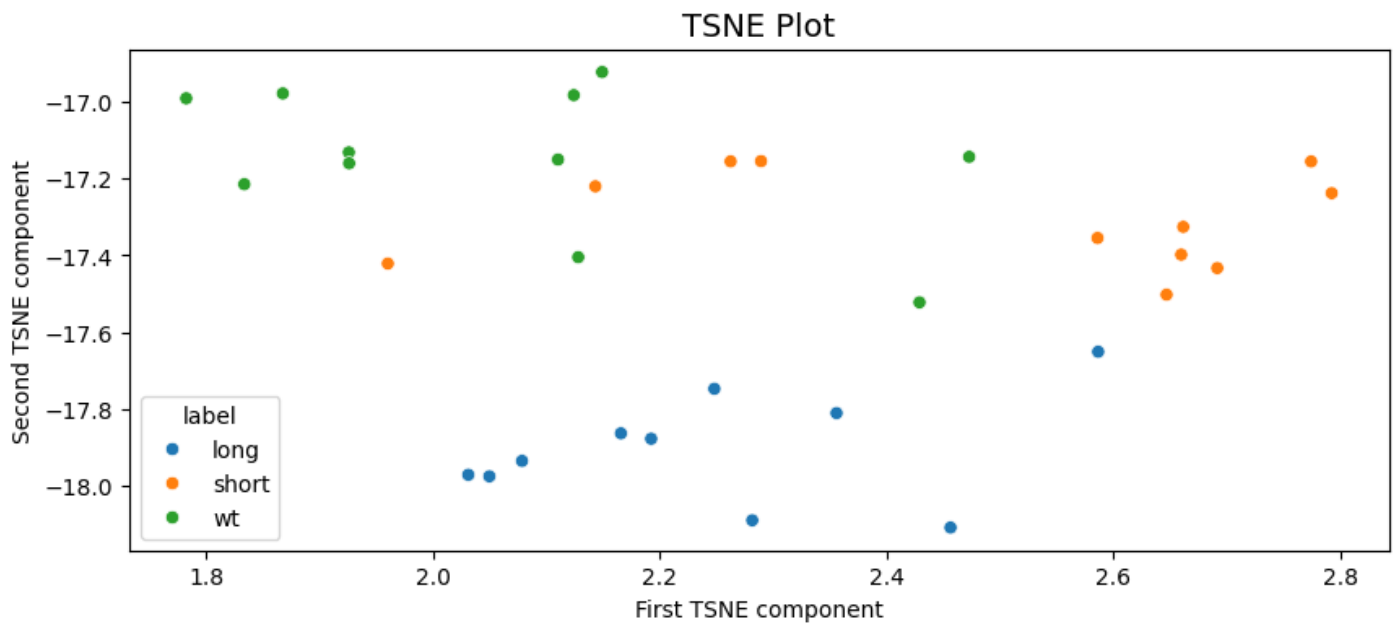
```

```
plt.figure(figsize=(10,4))
```

```
sns.scatterplot(data=ts_df,
                x="TS1",
                y="TS2",
                hue="label")

plt.title("TSNE Plot",
          fontsize=14)
plt.xlabel('First TSNE component',
          fontsize=10)
plt.ylabel('Second TSNE component',
          fontsize=10)
```

```
Text(0, 0.5, 'Second TSNE component')
```



A similar output to the PCA with long being clustered separately and some mixing of the short and WT labels in two clusters.

## 6. Classification

From the above plots it looks like the dataset is good for some classification attempts, we'll go through two of the

powerful but simple techniques SVM and Random Forest classifier.

```
# We need to load in a few more bits from sklearn
from sklearn.svm import SVC
from sklearn.model_selection import train_test_split
from sklearn.pipeline import Pipeline
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import accuracy_score, confusion_matrix, classification_report
from sklearn.preprocessing import LabelEncoder
from sklearn.ensemble import RandomForestClassifier
```

```
# Lets put our labels into variable y
y = c22_df['label'].tolist()
# And split our dataset into both a training set (70%) and testing set (30%)
# Stratify makes sure there is the same ratio of all groups in both train and test
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=12,
stratify = y)
```

```
# This function prints out accuracy and reports of the classifier
def print_score(clf, X, y):
    pred = clf.predict(X)
    clf_report = pd.DataFrame(classification_report(y, pred, output_dict=True))
    print(f"Accuracy Score: {accuracy_score(y, pred) * 100:.2f}%")
    print("_____")
    print("Classification Report:")
    print(clf_report)
    print("_____")
    print("Confusion matrix:")
    print(confusion_matrix(y, pred))
```

Support Vector Machines (SVM) is a machine learning algorithm that creates a decision boundary in the variable space that it thinks best separates your given groups. We'll kick off by just calling the most basic SVM, one that employs a linear kernel.

```
# Create the classifier and fit it to the data
svm_clf = SVC(kernel='linear')
svm_clf.fit(X_train, y_train)
# The train dataset score
print_score(svm_clf, X_train, y_train)
```

Accuracy Score: 95.45%

---

Classification Report:

	long	short	wt	accuracy	macro avg	weighted avg
precision	1.0	0.888889	1.000000	0.954545	0.962963	0.959596
recall	1.0	1.000000	0.857143	0.954545	0.952381	0.954545
f1-score	1.0	0.941176	0.923077	0.954545	0.954751	0.954134
support	7.0	8.000000	7.000000	0.954545	22.000000	22.000000

---

Confusion matrix:

```
[[7 0 0]
 [0 8 0]
 [0 1 6]]
```

```
# The test dataset score
print_score(svm_clf, X_test, y_test)
```

Accuracy Score: 60.00%

---

Classification Report:

	long	short	wt	accuracy	macro avg	weighted avg
precision	1.0	0.333333	0.5	0.6	0.611111	0.6
recall	1.0	0.333333	0.5	0.6	0.611111	0.6
f1-score	1.0	0.333333	0.5	0.6	0.611111	0.6
support	3.0	3.000000	4.0	0.6	10.000000	10.0

---

Confusion matrix:

```
[[3 0 0]
 [0 1 2]
 [0 2 2]]
```

The linear SVM performs very well on the training dataset (which is to be assumed), but has a poor score for the test

dataset. We could play around with the settings for SVM function manually or we can perform a grid search. A grid search takes a list of different inputs to the classifying function and runs all the combination, finding the one with the best score.

```
# First create a dictionary of the parameters you want to change and the list of variabels to apply
# We'll use all the kernels avaiable to us and a range of gamma and C
# Head to the sklearn wiki for information on tthe parameters
param_grid = {'C': [0.01, 0.1, 0.5, 1, 5, 10, 100],
              'gamma': [1, 0.75, 0.5, 0.25, 0.1, 0.01, 0.001, 'auto'],
              'kernel': ['rbf', 'poly', 'linear']}
# Now we setup the Grid search with the SVM, the parameters. CV is a resampling method that test and trains within the given dataset
grid_svm = GridSearchCV(SVC(), param_grid, verbose=True, cv=3)
```

```
# Fit the data to the gridsearch
grid_svm.fit(X_train, y_train)
params_svm = grid_svm.best_params_
print(f"Best params: {params_svm}")
```

Fitting 3 folds for each of 168 candidates, totalling 504 fits

```
Best params: {'C': 0.5, 'gamma': 0.25, 'kernel': 'poly'}
```

```
# Lets take the parameters and score then
svm_clf = SVC(**params_svm)
svm_clf.fit(X_train, y_train)
print('Training score:')
print_score(svm_clf, X_train, y_train)
print('Testing score:')
print_score(svm_clf, X_test, y_test)
```

```
Training score:
Accuracy Score: 86.36%
```

---

Classification Report:

	long	short	wt	accuracy	macro avg	weighted avg
precision	1.000000	0.727273	1.000000	0.863636	0.909091	0.900826
recall	0.714286	1.000000	0.857143	0.863636	0.857143	0.863636
f1-score	0.833333	0.842105	0.923077	0.863636	0.866172	0.865078
support	7.000000	8.000000	7.000000	0.863636	22.000000	22.000000

---

Confusion matrix:

```
[[5 2 0]
 [0 8 0]
 [0 1 6]]
```

Testing score:

Accuracy Score: 80.00%

---

Classification Report:

	long	short	wt	accuracy	macro avg	weighted avg
precision	1.0	0.60	1.000000	0.8	0.866667	0.880000
recall	1.0	1.00	0.500000	0.8	0.833333	0.800000
f1-score	1.0	0.75	0.666667	0.8	0.805556	0.791667
support	3.0	3.00	4.000000	0.8	10.000000	10.000000

---

Confusion matrix:

```
[[3 0 0]
 [0 3 0]
 [0 2 2]]
```

The grid searched parameters perform worse on the trained dataset, but better on the test set. Meaning the it's not as overfitted to the training set!

Lets have a look at an SVM when applied to the PCA results from above to gain a visual understanding of what's going on

```
# We need to encode the labels as numbers to colour code them with the plotting function below
le = LabelEncoder()
le.fit(['short', 'wt', 'long'])
y2 = le.transform(pca_df.label.to_list())
```

```

# We'll take the values for the PCA dataframe, and the set the labels a yp
Xp = pca_df.values[:, :2]
yp = y2

# Creates a grid for the plot points to applied to
def make_meshgrid(x, y, h=.02):
    x_min, x_max = x.min() - 0.1, x.max() +0.1
    y_min, y_max = y.min() - 0.1, y.max() + 0.1
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))
    return xx, yy

# This function plots the different areas the SVM decides for each label in the variable space
def plot_contours(ax, clf, xx, yy, **params):
    Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)
    out = ax.contourf(xx, yy, Z, **params)
    return out

# Call a grid search on this PCA values and call the best fit model
grid.fit(Xp, yp)
best_params = grid.best_params_
model = SVC(**best_params)
clf = model.fit(Xp, yp)

fig, ax = plt.subplots()
title = ('Decision surface of the best fit SVM')
# Set-up grid for plotting.
X0, X1 = Xp[:, 0], Xp[:, 1]
xx, yy = make_meshgrid(X0, X1)

# Plot the svm areas and variable points
plot_contours(ax, clf, xx, yy, cmap = 'coolwarm', alpha=0.8)
ax.scatter(X0, X1, c=yp, s=20, cmap = 'coolwarm', edgecolors='k')
ax.set_ylabel('PCA 2')
ax.set_xlabel('PCA 1')
ax.set_xticks(())
ax.set_yticks(())
ax.set_title(title)
plt.show()

```

```

-----
NameError                                Traceback (most recent call last)

Cell In[40], line 20
     16     out = ax.contourf(xx, yy, Z, **params)
     17     return out
     18
     19 # Call a grid search on this PCA values and call the best fit model
--> 20 grid.fit(Xp, yp)
     21 best_params = grid.best_params_
     22 model = SVC(**best_params)
     23 clf = model.fit(Xp, yp)

NameError: name 'grid' is not defined

```

This visualises nicely the problem the SVM is having with classifying at 100%, it's the points we noticed above that are WT but have short sleeping like phenotype

Next we'll look at random forest classifiers (RFC). RFC's are a collection of tree classifiers, each tree classifier works on a subset of the data, making decisions at each leaf to bag the data into one of the labels. On their own they can often overfit to the training dataset, however as a forest and added randomness this can be accounted for. Random forests often perform the best for a wide range of datasets.

```

# Lets initialise the classifier and create a grid search for the off
# Head to the sklearn wiki to understand the parameters
rfc = RandomForestClassifier(random_state=42)
param_grid = {'n_estimators': [100, 200, 300, 500],
              'max_features': ['sqrt', 'log2'],
              'max_depth' : [4,5,6,7,8,10],
              'criterion' :['gini', 'entropy']}

```

```
grid_rfc = GridSearchCV(rfc, param_grid, refit=True, verbose=1, cv=3)
```

```
# Call and fit the grid search for the RFC
grid_rfc.fit(X_train, y_train)
params_rfc = grid_rfc.best_params_
print(f"Best params (RFC): {params_rfc}")
```

Fitting 3 folds for each of 96 candidates, totalling 288 fits

```
Best params (RFC): {'criterion': 'gini', 'max_depth': 4, 'max_features': 'sqrt',
'n_estimators': 300}
```

```
# Lets see how it's done
rfc_clf = RandomForestClassifier(**params_rfc, random_state = 42)
rfc_clf.fit(X_train, y_train)
print('Scores for the trained dataset:')
print_score(rfc_clf, X_train, y_train)
print('Scores for the test dataset:')
print_score(rfc_clf, X_test, y_test)
```

Scores for the trained dataset:

Accuracy Score: 100.00%

---

Classification Report:

	long	short	wt	accuracy	macro avg	weighted avg
precision	1.0	1.0	1.0	1.0	1.0	1.0
recall	1.0	1.0	1.0	1.0	1.0	1.0
f1-score	1.0	1.0	1.0	1.0	1.0	1.0
support	7.0	8.0	7.0	1.0	22.0	22.0

---

Confusion matrix:

```
[[7 0 0]
 [0 8 0]
 [0 0 7]]
```

Scores for the test dataset:

Accuracy Score: 60.00%

---

Classification Report:

	long	short	wt	accuracy	macro avg	weighted avg
--	------	-------	----	----------	-----------	--------------

```
precision    1.0  0.333333  0.5         0.6  0.611111         0.6
recall      1.0  0.333333  0.5         0.6  0.611111         0.6
f1-score    1.0  0.333333  0.5         0.6  0.611111         0.6
support     3.0  3.000000  4.0         0.6  10.000000        10.0
```

---

Confusion matrix:

```
[[3 0 0]
 [0 1 2]
 [0 2 2]]
```

Sadly it's not performed as well as the best SVM. We could play around with more parameters, but for now it looks like the SVM score of 80% is the best we'll get (which is still pretty good).

If you'd like to perform more tests look online for tutorials about the other classifiers in the sklearn ecosystem. You can also head to the HCTSA tutorial page and look to copy some of their techniques from matlab to python:

<https://hctsa-users.gitbook.io/hctsa-manual/>

---

Revision #2

Created 2026-04-22 10:29:48 UTC by Giorgio Gilestro

Updated 2026-04-22 10:33:24 UTC by Giorgio Gilestro