MIRS Lab guides

Dr. Mauro Pazmino

2024-10-22

Table of contents

Preface **4**

1 How to use our scripts for mid infrared spectroscopy **5**

1.1 Opus dei 5
1.2 Bad blood 5
1.3 How to use the scripts 6
1.3.1 Using only jupyter notebook 6
1.3.2 Using a jupyter notebook and the .py files approach 6
1.4 Examples on how to use the opus_dei.py and bad_blood.py 7
1.5 Errors using bad blood 10

2 Visualization of spectroscopy data **11**

2.1 Basic plot with defaults 11
2.2 Improving the defaults 12
2.3 Plotting the means of different classes 14
2.4 Plotting all samples of different classes 15

3 Machine learning pipelines **17**

3.1 Basic exploratory machine learning analysis 17
3.1.1 Explore different models 17
3.1.2 Model optimization 19
3.1.3 Test the final model 20
3.1.4 How can we check what features are important? 21
3.1.5 Permutation importance 24
3.1.6 Permutation importance using RF 26
3.2 Nested Cross validation 30
3.3 Nested cross validation plus validation on unseen data 39
3.3.1 Set plot aesthetics 40
3.3.2 Defining the fuctions 40
3.3.3 Split the data 45
3.3.4 Setting the parameters of nested cross validation 45
3.3.5 Setting the parameters of nested cross validation 46
3.3.6 Run the loop 46

4 Deep Learning pipelines 51
4.1 A basic artificial neural network for species classification . . . 51
4.1.1 Load and preprocess the data . . . 51
4.1.2 Building the model . . . 56
4.1.3 Compiling the model . . . 56
4.1.4 Training the model . . . 57
4.1.5 Evaluate with unseen data . . . 59
4.1.6 Save your model . . . 61
4.2 A basic artificial neural network for multiclass age classification . . . 62
4.2.1 Encoding labels . . . 63
4.2.2 Building the model . . . 64
4.2.3 Evaluate the model with useen data . . . 65
4.2.4 Reverse the encoding . . . 65
4.3 Evaluate your model using crossvalidation. . . . 68
4.3.1 Define the cross-validator . . . 68
4.3.2 Cross validation process . . . 69
4.3.3 Plotting the mean accuracy and standard deviation . . . 75
4.4 Tuning a deep learning model . . . 77

5 XGboost for spectroscopy 78
5.1 XGboost: A primer . . . 78
5.1.1 Encoding our labels . . . 78
5.1.2 Importing the model . . . 79
5.1.3 Train and test . . . 79
5.1.4 Feature importance . . . 81

6 A closer look into spectral quality 82
6.1 Low intensity . . . 83
6.2 Atmospheric intrusion . . . 83
6.3 Anvil distortion . . . 84

References 85

Preface

This document is a series of guides to help students that are working with infrared spectroscopy in the context of vector surveillance. Here, there are information about to the use of our scripts to extract data, clean it, assembling data sets for machine/deep learning as well as tutorials on how to plot data, machine/deep learning pipelines.

This guide is based on the follwing published work:

- González Jiménez, Mario, et al. "Prediction of mosquito species and population age structure using mid-infrared spectroscopy and supervised machine learning." Wellcome open research 4 (2019).(González Jiménez et al. 2019)
- Siria, Doreen J., et al. "Rapid age-grading and species identification of natural mosquitoes for malaria surveillance." Nature Communications 13.1 (2022): 1501. (Siria et al. 2022).
- Mwanga, E.P., Siria, D.J., Mitton, J. et al. Using transfer learning and dimensionality reduction techniques to improve generalisability of machine-learning predictions of mosquito ages from mid-infrared spectra. BMC Bioinformatics 24, 11 (2023). (Mwanga et al. 2023)
- Pazmiño-Betancourth, Mauro, et al. "Evaluation of diffuse reflectance spectroscopy for predicting age, species, and cuticular resistance of Anopheles gambiae sl under laboratory conditions." Scientific Reports 13.1 (2023): 18499.(Pazmiño-Betancourth et al. 2023)
- Pazmiño-Betancourth, Mauro, et al. "Advancing age grading techniques for Glossina morsitans morsitans, vectors of African trypanosomiasis, through mid-infrared spectroscopy and machine learning." Biology Methods and Protocols 9.1 (2024): bpae058. (Pazmiño-Betancourth et al. 2024)

if you have any questions you can reach me or Ivan Casas

1 How to use our scripts for mid infrared spectroscopy

This is a quick tutorial on how to use the scripts for decoding files from the FTIR machine and to clean up and assembling all the files into a csv file.

! Important

These guides assume you have some knowledge about python such as installing/importing packages, building functions, working with jupyter notebooks, etc.

The scripts are:

- opus_dei
- bad_blood

1.1 Opus dei

This script takes the files from the FTIR machine that are proprietary (you can only read them with OPUS software) and transform them into .dpt (data point table) files and .mzz files.

Dpt files are basically a two column file with one column will have the wavenumbers value and the second column will have the absorbance. It can be opened in excel, notes, etc.

Mzz files are compressed files especially created for this project. They contain only the absorbance and can be opened by bad_blood script.

1.2 Bad blood

This script reads all your files either as .dpt or .mzz and assess the quality of each file (low intensity, atmospheric interference, and distortion by the anvil). Then, it will create a table with all the absorbance, and metadata that you can use for your machine learning analysis.

1.3 How to use the scripts

There are two approaches (maybe three) on how you can use them in your workflow. These are:

1. Using the jupyter notebook file only
2. Using a combination of jupyter notebook and the .py files

1.3.1 Using only jupyter notebook

This is self explanatory. You just need to copy the jupyter notebook into your working folder. Open it with jupyter or jupyterlab or vscode and follow the instructions.

1.3.2 Using a jupyter notebook and the .py files approach

This is the approach for intermediate/advance users. Combines the flexibility of jupyter notebooks plus the convenience of having all your functions in a separate file. This improves readability, maintainability of your code (and it looks more slick!) For this you will need to have a specific folder structure. Something like this:

In the folder called source code (src), you will put the bad_blood.py and opus_dei.py files. Then, you will create your jupyter notebook for your analyses. You need to include two important pieces of code.

First:

```
%load_ext autoreload
%autoreload 2
```

This code you will only need to run it once (at the beginning of your session). This allows you to change or improve your functions from the source code folder and they will be updated automatically.

The second code block you will need is this:

```
sys.path.append('/your/path/toyour/sourcecode/src’)
```

You need to specify the path where bad blood and opus dei are located, so jupyter knows where to search when you import those functions. This can be added in the cell where you import all your packages.

Tip

Try to use a IDE such as Visual Studio Code from the beginning.

1.4 Examples on how to use the opus_dei.py and bad_blood.py

To run opus dei and bad blood you need the following packages:

You need to set up your jupyter notebook (this also work if you are only using jupyter)

```
# setting up the path to src folder
import sys
sys.path.append("/Users/mauropazmino/Documents/University/Workhops/How to use the scripts/

# the packages that you will use
import numpy
import pandas

# importing opus dei and bad blood functions
from bad_blood import bad_blood_v3_real
from opus_dei import opus_dei
```

Once you have imported your modules. You need to run opus_dei function to decode the files into readable format.

The opus dei function requires two parameters: the folder where the raw files are located and the folder where you want to save the new files.

```
raw_data_folder = "./data/raw_data"

processed_data_folder = "./data/processed_data"

# run the function
opus_dei(raw_data_folder, processed_data_folder)
```

```
Your files have been proceesed!!!
```

Once the files has been decoded, you will get a message.

After that, you can run the bad blood function to the processed files.

The bad blood function requires two parameters, one is the path where the processed files are located and two, the name of the assembled datafile. Moreover, this function requires all the files have the same file length.In our lab, we have the following filename convention:

- XX-XX-XX-XX-XX.dpt

where each part of the file that is separate by a hyphen represents some metadata.For example in the following file: UV-AG-F-NOU-R1.0.dpt

UV means UV experiment, AG means Anopheles gambiae, F meeans female, etc. It does not matter how many sections you have in your file name (2 from 100), as long as all your files have the same number of sections. If one file has less or more, the function will give you an error and it will indicate which files need to be fixed. You can do that on the .mzz files but it would be better if you do it in your raw files and then run opus dei again.

> ! File name length is important!
>
> Always check that your files have the same length.

Now you run the function bad_blood

```
processed_data_folder = "./data/processed_data"
name_data = 'lab_guide_sample'

bad_blood_v3_real(processed_data_folder,name_data)
```

```
  0%|          | 0/9 [00:00<?, ?it/s]

100%|      | 9/9 [00:00<00:00, 392.07it/s]




Assessing spectra with low quality..
    0 spectra have been discarded because their low intensity

Checking for abnormal background...
    0 spectra have been discarded because they were distorted by the anvil

Checking for spectra with atmospheric interferences..
    0 spectra have been discarded because have atmospheric interferences
```

```
Cat 1 - Options:
----------------
      UV
Cat 2 - Options:
----------------
      AG
Cat 3 - Options:
----------------
      F
Cat 4 - Options:
----------------
      NOU
Cat 5 - Options:
----------------
      R1
Cat 6 - Options:
----------------
      6
      2
      8
      4
      1
      5
      3
      7
      0

  0%|          | 0/9 [00:00<?, ?it/s]

100%|     | 9/9 [00:00<00:00, 509.33it/s]

This last process has lasted 0.019 s
The new matrix contains 9 spectra.
Your new matrix MIRS_lab_guide_sample_20241108 is in the folder data
```

As you can see, bad blood will read all your files and will do a quality control on each one of them. If a file does not pass the quality control, it will be discarded. After that, it will put all you files together with the metadata tied to them in a nice table format. The name of the final data set will include the date when it was created for future reference.

1.5 Errors using bad blood

❗ Errors when using bad blood

Bad blood will give the following error if the folder where it reads the files does not not have any mzz or dpt files on it.

```
UnboundLocalError: cannot access local variable 'tmp2'
where it is not associated with a value
```

If you want to know more what bad blood does, you can check Chapter 6 for more information

2 Visualization of spectroscopy data

This is a quick tutorial on how to plot and make publication-quality figures from infrared spectroscopy data using Python. We are using matplotlib, pandas, numpy and seaborn for this tutorial. Check the requierements file. The data used in this tutorial is from a small experiment where we collected spectra from mosquito thoraces using a Attenuated Total Reflection. There are two classes, mosquitoes exposed to UV and non exposed to UV.

2.1 Basic plot with defaults

First, we need to import the data that we are going to use. The data will be arranged with samples as rows and wavenumbers and categories as a columns.

```
# import packages
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sn
```

```
data = pd.read_csv("./data/UV_pilot.csv")
data.head()
```

	Specie	Sex	Exposed	Reeplicate	ID	4006	4004	4002	4000	3998	...	415
0	AG	F	YES	R1	2	0.00477	0.00462	0.00448	0.00434	0.00439	...	0.33421
1	AG	F	YES	R1	3	0.00596	0.00631	0.00629	0.00621	0.00623	...	0.32344
2	AG	F	YES	R1	8	0.00542	0.00513	0.00511	0.00517	0.00537	...	0.33420
3	AG	F	NOU	R1	1	0.00681	0.00690	0.00695	0.00683	0.00666	...	0.29993
4	AG	F	NOU	R1	3	0.00543	0.00552	0.00556	0.00555	0.00548	...	0.29425

We select the wavenumbers and their absorbance values and we call this "X". We also select the categorical values we want to plot, and call it "y_labels"

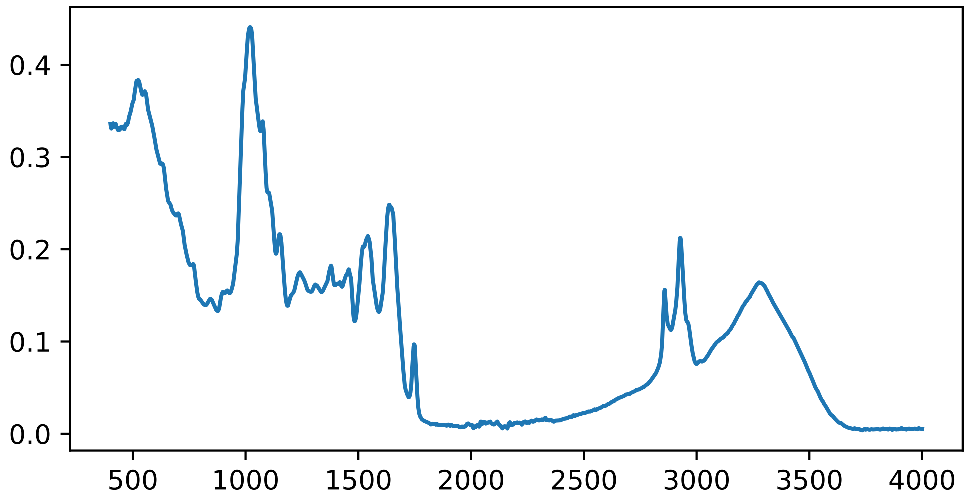
```
X = data.loc[:,"4000":'401']
y_labels = data.loc[:,"Exposed"]
```

To plot a line, we need x and y values. The values of x will be the specific wavenumbers (which are the names of the columns of X). We extract the names of the columns of X and put them on a list as a integers.

```
wnLabels = X.columns.values
waveNums = [int(x) for x in wnLabels]
```

Now that we have teh values of x, we need the absorbance of a sample. We can select any row from X.

```
fig, ax = plt.subplots(figsize=(6,3))
ax.plot(waveNums, X.iloc[2,:])
```

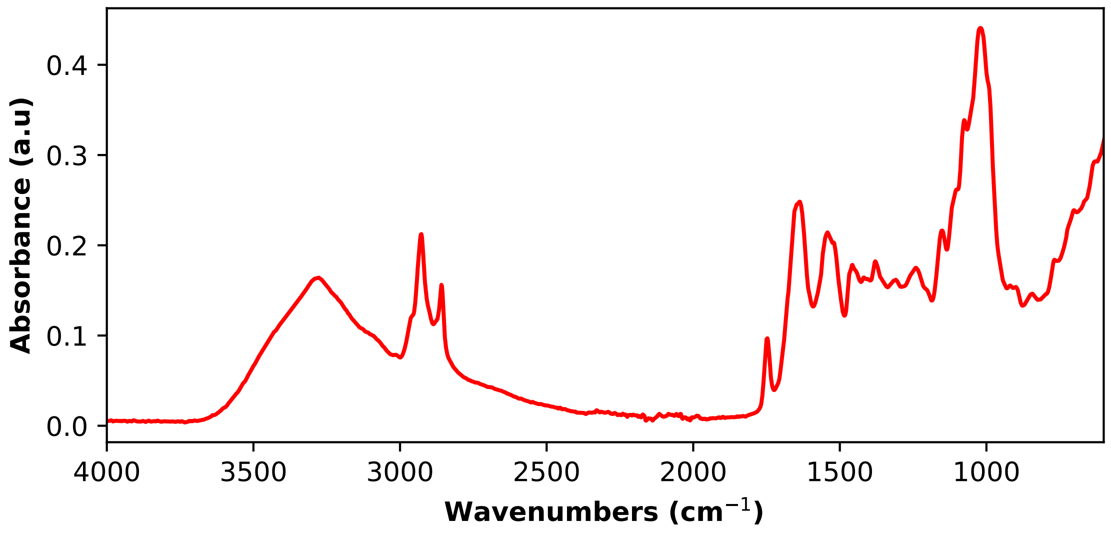


2.2 Improving the defaults

The defaults of matplotlib or seaborn are not that great. Increasing the linewidth and font size, choosing another color and add axis labels can help with the overall quality and readability of the plot. Moreover, infrared spectral data is plot from 4000 to 400 cm^{-1}. So, we need to reverse the x-axis.

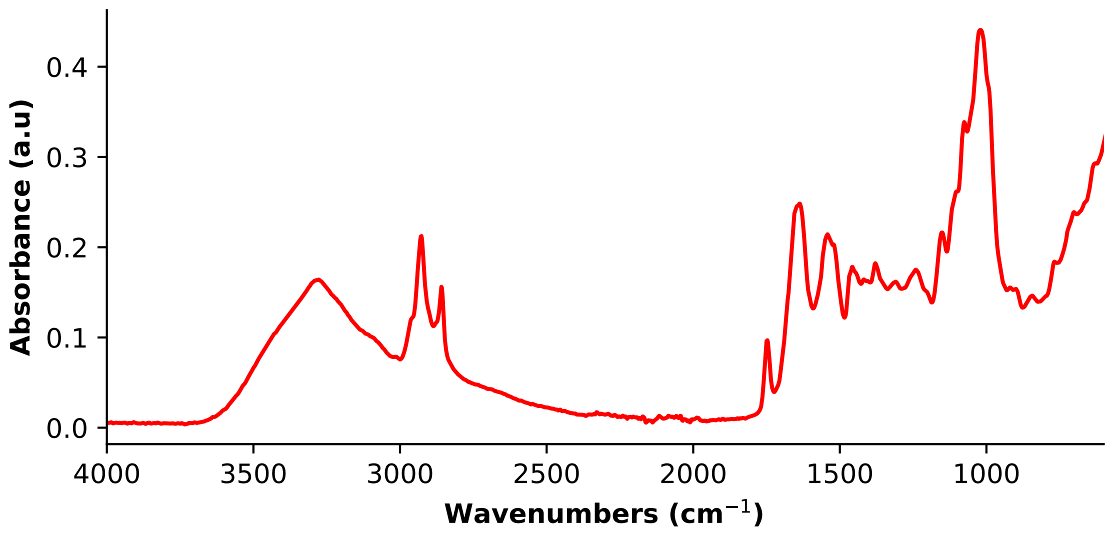
```
fig, ax = plt.subplots(figsize=(6,3), tight_layout=True)
ax.plot(waveNums,X.iloc[2,:],color='r',linewidth=1.5)

ax.set_xlim(4000, 600)
ax.set_ylabel("Absorbance (a.u)", fontweight='bold')
ax.set_xlabel('Wavenumbers (cm$^{-1}$)', fontweight='bold')
```

```
Text(0.5, 0, 'Wavenumbers (cm$^{-1}$)')
```



We can choose different aesthetics for the final plot like removing the spines.

```
fig, ax = plt.subplots(figsize=(6,3), tight_layout=True)
ax.plot(waveNums,X.iloc[2,:],color='r',linewidth=1.5)

ax.set_xlim(4000, 600)
ax.set_ylabel("Absorbance (a.u)", fontweight='bold')
ax.set_xlabel('Wavenumbers (cm$^{-1}$)', fontweight='bold')
ax.spines.right.set_visible(False)
ax.spines.top.set_visible(False)
```



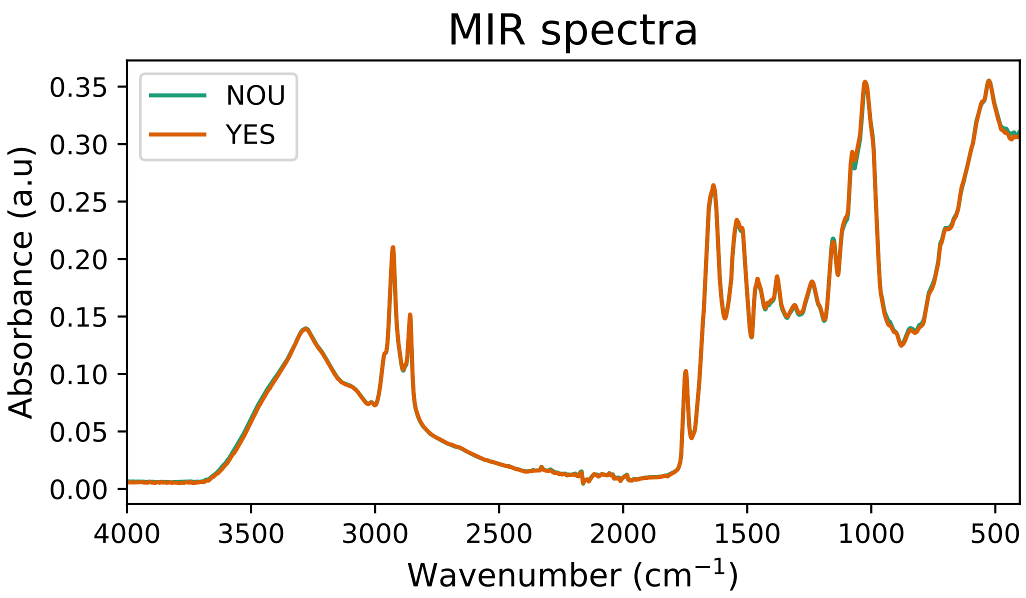
2.3 Plotting the means of different classes

To observe differences between classes, you might want to plot the mean spectra of each class you are assessing. The issue here is that the data is presented as rows, which complicates the plotting process since matplotlib like the samples in columns rather than rows.

The code is straightforward, and it works for any number of classes (as long as the number of colours is the same of greater than the number of classes)

```
# color hex codes can be used, as well as common color names. You can get more codes at ht

colors = ['#1b9e77','#d95f02','#7570b3','#e7298a','#66a61e']

# figure
fig, ax = plt.subplots(figsize=(6,3))


for i, c in zip(np.unique(y_labels), colors):
    sn.lineplot(x=waveNums, y=np.mean(X[y_labels == i], axis=0), label=i, color=c)
plt.legend()
ax.set_xlim(4000,401)
ax.set_title("MIR spectra",fontsize=16)
ax.set_xlabel("Wavenumber (cm$^{-1}$)",fontsize=12)
ax.set_ylabel("Absorbance (a.u)",fontsize=12)
```

```
Text(0, 0.5, 'Absorbance (a.u)')
```



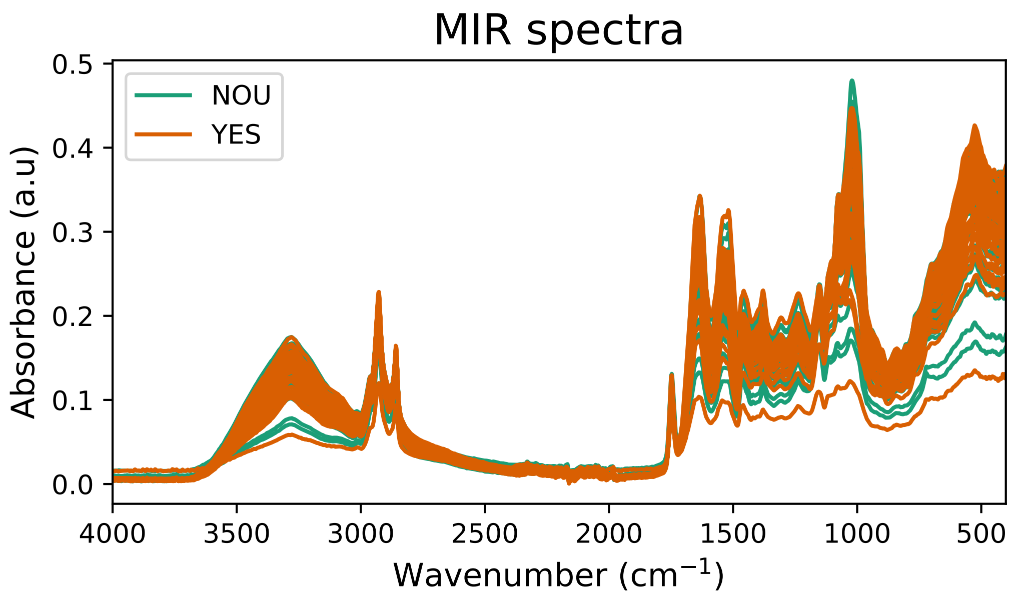
2.4 Plotting all samples of different classes

What if you want to plot not the means but all the samples? For this, we use the same approach as before, but instead of selecting all the samples and plot the mean, we select each sample and plot it. We use itertuples to go through them each row and plot it with the assigned colour. It might be a little bit slow if you have thousands of rows.

```python
fig, ax = plt.subplots(figsize=(6,3))

# Iterates through each row and compare it to the label, and plot it.
for i, c in zip(np.unique(y_labels), colors):
    for row in X.loc[y_labels == i].itertuples(index=False):
        ax.plot(waveNums, row, color=c, label=i)

# Legend creation
handles, labels_2 = ax.get_legend_handles_labels()
newLabels, newHandles = [], []
for handle, label_1 in zip(handles, labels_2):
    if label_1 not in newLabels:
        newLabels.append(label_1)
```

```
            newHandles.append(handle)

    ax.legend(newHandles, newLabels)
    ax.set_xlim(4000,401)
    ax.set_title("MIR spectra",fontsize=16)
    ax.set_xlabel("Wavenumber (cm$^{-1}$)",fontsize=12)
    ax.set_ylabel("Absorbance (a.u)",fontsize=12)
```

```
Text(0, 0.5, 'Absorbance (a.u)')
```



3 Machine learning pipelines

In this section, we are going to describe some of the pipelines that our group uses as a starting point in the analysis of infrared data. If you want to explore deeper about machine learning, models, optimization, etc, a really good source is the scikit-learn user guide

3.1 Basic exploratory machine learning analysis

This is a pipeline that can be used to compare different models and decide which one is the best for your case and further optimize it and test it on our test data set.

3.1.1 Explore different models

First, you need to import your data, separate features (absorbances) and the target variable, and split the data set into train and test set.

```
# Import data
df = pd.read_csv("./data/UV_pilot.csv")

# Split features and target
X = df.loc[:,"4000":"403"]
y = df.loc[:,"Exposed"]

# Split into train and test sets

X_train, X_test, y_train, y_test = train_test_split(X, y, shuffle=True, stratify=y, test_s
```

❗ Important

Be sure you stratify your split by using the `stratify` parameter. This is to make sure the different classes of the target varibale are equality distribute in the split

You need to create a dictionary of the models you want to test. For this example, we are going to use 3 models: Logistic regression, Support Vector Machine, and Random Forest but you can add more models if needed.

```
model_dict = {
    "LR": LogisticRegression(),
    "SVM": SVC(),
    "RF": RandomForestClassifier(),
}
```

It is good practice to use pipelines to avoid data leakage especially, if we are using preprocessing algorithms in our data. Usually, we apply standard scaling, so for this, we will build a pipeline that includes the preprocessing algorithm and the model. Click here if you want to learn more about pipelines and how to use them.

Then, we create a for loop which is going to iterate between each model, do a cross validation with our specified pipeline, append the results into a list and print a message with the accuracy and the standard deviation of each model.

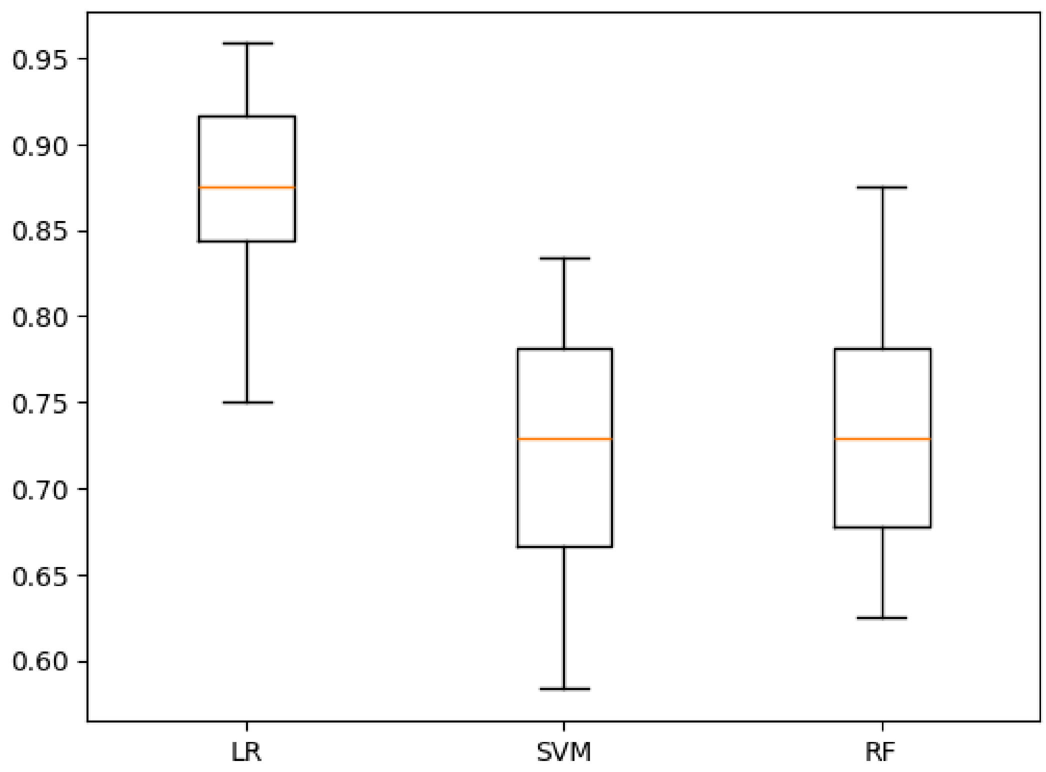
```
results = []
names = []
sss = StratifiedShuffleSplit(n_splits=10, test_size=0.3, random_state=7)

for key, model in model_dict.items():
    pipe = Pipeline([('scaler', StandardScaler()), ('model', model)])

    cv_results = cross_val_score(pipe, X_train, y_train, cv=sss, scoring="accuracy")
    results.append(cv_results)
    names.append(key)
    print(f'Accuracy using {key} is {cv_results.mean():.2f} ± {cv_results.std():.2f}')
```

```
Accuracy using LR is 0.87 ± 0.07
Accuracy using SVM is 0.72 ± 0.08
Accuracy using RF is 0.74 ± 0.07
```

We can examine our results more with a plot

```
fig, ax = plt.subplots()
ax.boxplot(results)
ax.set_xticklabels(names)
```

```
[Text(1, 0, 'LR'), Text(2, 0, 'SVM'), Text(3, 0, 'RF')]
```



From the plot, we can see that logistic regression has the highest accuracy. So, we choose it as our model to further optimize it.

3.1.2 Model optimization

Model optimization consist on finding the best combination of our model's hyperparameter values that will give us the best accuracy. For this, we need to know which hyperparameters can be change, what values they accept and how the affect the model.

For logistic regression we can try different values of C, the type of solver it uses and the type of penalty. We create a dictionary with the hyperparameters and the different values we cant to try.

```
# create a dictionary with different hyperparameters and the values you want to test

param_grid = {'model__penalty': ['l1', 'l2'],
                  'model__C': [0.01, 0.1, 1, 10],
                  'model__solver': ['liblinear', 'saga']}
```

```
# create the new pipleine with the model we choose to optimize
pipe = Pipeline([('scaler', StandardScaler()), ('model', LogisticRegression(max_iter=10000
```

Sklearn in-built function **GridsearchCV** will go over each of the possible combinations between the different values of each hyperparameter and calculate the accuracy of each combination. Then, it will chose the model with the best accuracy and the parameters that uses.

```
search = GridSearchCV(pipe, param_grid, n_jobs=2)
search.fit(X_train, y_train)
print(f"Best parameter (CV score={search.best_score_:.2f})")
print(search.best_params_)
```

```
Best parameter (CV score=0.91)
{'model__C': 1, 'model__penalty': 'l1', 'model__solver': 'saga'}
```

Here, we can see that the best parameters with a cross-validation value of 0.91 are C=10, penalty=l1, solver=liblinear.You can select the best pipleine from the object using `.best_estimator_` atrribute.

```
search.best_estimator_
```

```
Pipeline(steps=[('scaler', StandardScaler()),
                ('model',
                 LogisticRegression(C=1, max_iter=10000, penalty='l1',
                                    solver='saga'))])
```

! Important

Model optimization can be a extremely long process. Be mindful of the hyperparameters you want to test. This will require you to really study the model and make sense of the values you are going to use. Again, scikit-learn documentation is extremely helpful!

3.1.3 Test the final model

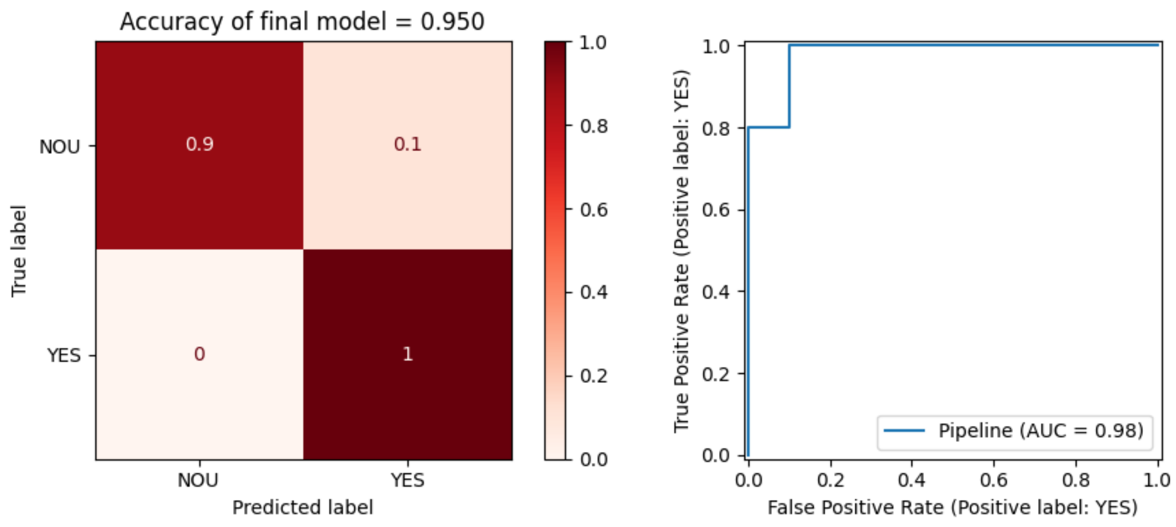
The final optimized model (or in this case pipeline) is tested on the unseen data you split at the beginning of the analysis. Fit the optimized model to your train data set and calculate the predicted values

```
optimized_model = search.best_estimator_
optimized_model.fit(X_train, y_train)
y_pred = optimized_model.predict(X_test)
```

Your final score, confusion matrix and auc-roc curve

```
# Accuracy and confusion  matrix

fig, (ax, ax2) = plt.subplots(1,2, figsize=(10,4), tight_layout=True)


ConfusionMatrixDisplay.from_predictions(y_test, y_pred, normalize='true', cmap='Reds',ax=a

RocCurveDisplay.from_estimator(optimized_model, X_test, y_test, ax=ax2)


title1 = (f'Accuracy of final model = {accuracy_score(y_test, y_pred):.3f}')
ax.set_title(title1)
```

```
Text(0.5, 1.0, 'Accuracy of final model = 0.950')
```



3.1.4 How can we check what features are important?

This is a hot topic, at least for me. For various reasons: correlation of the features, the model that you use, misleading results, etc. A way of have the feeling about what the model is learning is to look at the coefficients. Some models will have feature importance as an attribute (random forests and XGboost) as well SVC with a non-linear kernel. Whatever you will use, this is how you access them.

For this example, I will use the dataset but only suing 14 wavenumbers from González Jiménez et al. (2019).

```
# Import data
df = pd.read_csv("./data/UV_pilot.csv")

wavenumbers = ['3856', '3401', '3275', '2923', '2859', '1902', '1745', '1636', '1538', '14


# Split features and target
X = df[wavenumbers]
y = df.loc[:,"Exposed"]

# Split into train and test sets

X_train, X_test, y_train, y_test = train_test_split(X, y, shuffle=True, stratify=y, test_s
```

I have already selected an optimized pipeline.

```
optimized_model = search.best_estimator_
optimized_model.fit(X_train, y_train)
```

```
Pipeline(steps=[('scaler', StandardScaler()),
                ('model',
                 LogisticRegression(C=1, max_iter=10000, penalty='l1',
                                    solver='saga'))])
```

You can access the attributes of each part of the pipeline using `named_steps` function.

```
# accessing the model coefficients
optimized_model.named_steps['model'].coef_
```

```
array([[-0.73180761, -1.13038731,  0.        ,  0.        ,  0.        ,
         0.        , -0.02957209,  0.        ,  0.        ,  0.        ,
         0.01832993, -1.26561288,  1.76822707,  0.        ,  0.        ,
         0.        ,  0.        ]])
```

The number of coefficients will be equal to the number of features. They will be ordered the same as you input as targets.

You can create a dataframe with the

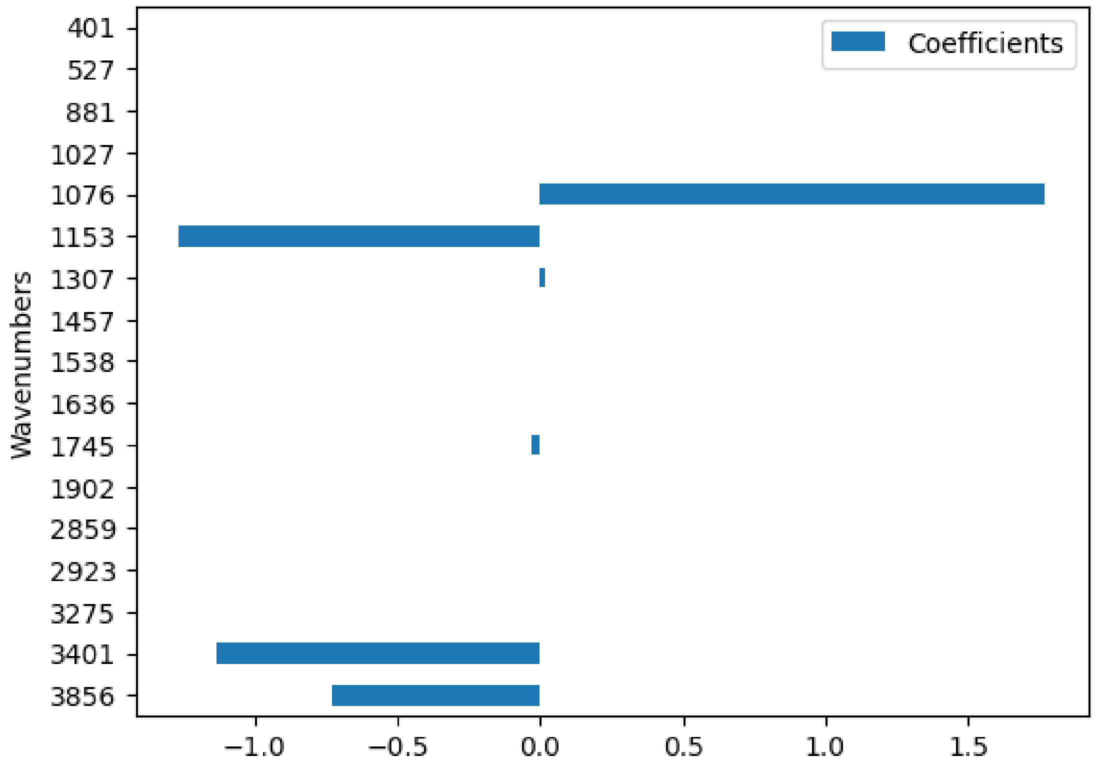
```
feature_importance = pd.DataFrame()
feature_importance['Wavenumbers'] = wavenumbers
feature_importance['Coefficients'] = optimized_model.named_steps['model'].coef_.T
feature_importance['Wavenumbers'] = feature_importance['Wavenumbers'].astype('category')
feature_importance
```

	Wavenumbers	Coefficients
0	3856	-0.731808
1	3401	-1.130387
2	3275	0.000000
3	2923	0.000000
4	2859	0.000000
5	1902	0.000000
6	1745	-0.029572
7	1636	0.000000
8	1538	0.000000
9	1457	0.000000
10	1307	0.018330
11	1153	-1.265613
12	1076	1.768227
13	1027	0.000000
14	881	0.000000
15	527	0.000000
16	401	0.000000

Finally, you plot as a barplot. The values of the coefficients can be positive or negative. If they are positive they push the decisiion to the negative class.

```
feature_importance.plot.barh(x='Wavenumbers', y='Coefficients')
```

```
<Axes: ylabel='Wavenumbers'>
```



And there you have it. If your model has a feature importance attribute, the same principle applies. There are other approaches such as permutation importance which can be used when you are using discrete numbers of features.

3.1.5 Permutation importance

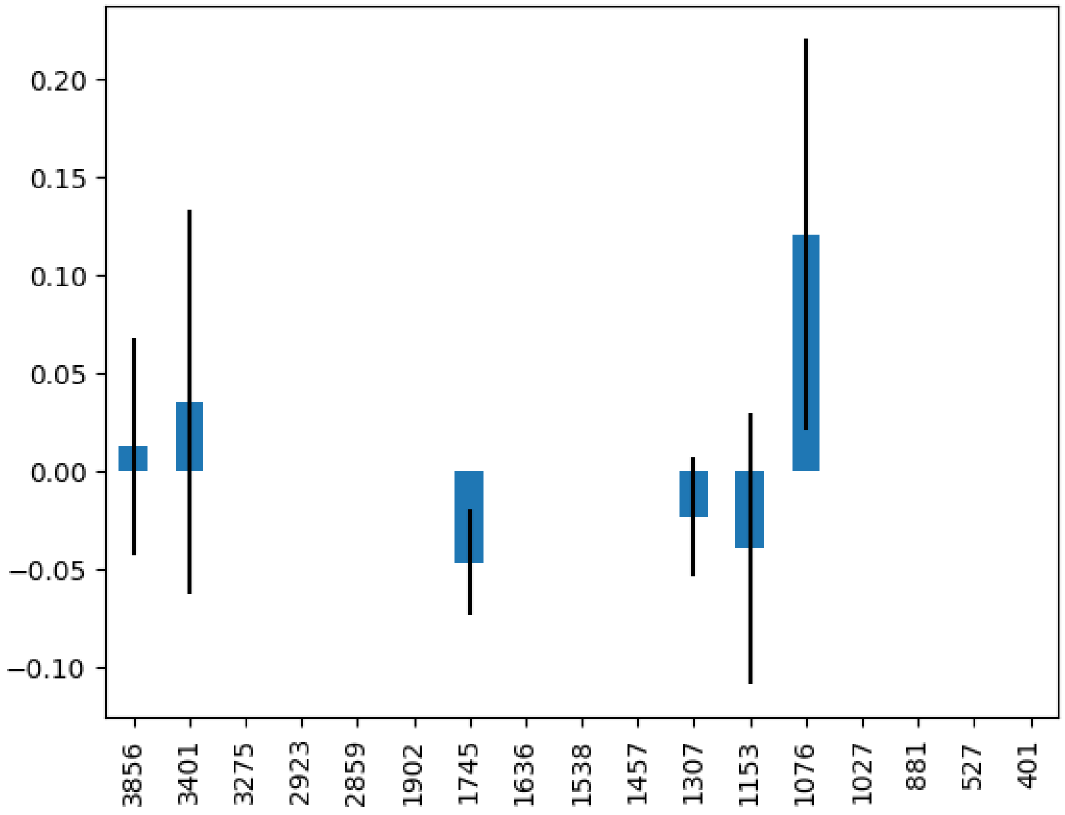
Sklearn has a Permutation importance class that you can use. Basically, a baseline accuracy is calculated with an unseen test set. Then, a feature is selected and permutated, the new accuracy is calculate and the permutation importance is given by the diffferentece between the baseline and the permutated accuracy. You repeat the process n times and you can then check the values and see what feature caused the most decrease in accuracy.

```
# permutation importance
from sklearn.inspection import permutation_importance
r = permutation_importance(optimized_model, X_test, y_test,
                           n_repeats=50,
                           random_state=0, scoring='accuracy')
```

You can plot the mean with the standard deviation pretty easily

```
# create a series of the importances mean and add the names of the wavenumber values
forest_importances = pd.Series(r.importances_mean, index=X.columns.values)

# Barplot with sd bars
forest_importances.plot.bar(yerr=r.importances_std)
```

```
<Axes: >
```



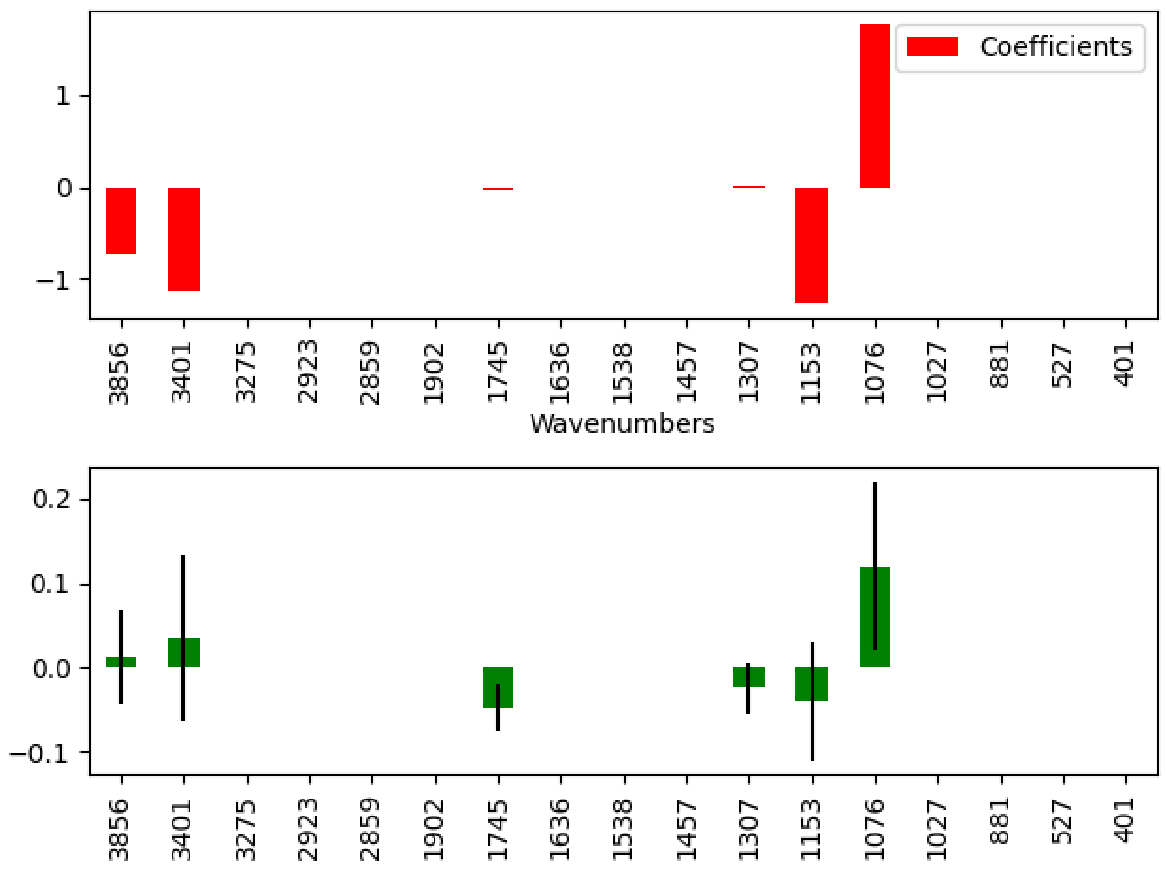
Here, we can see how the absorbance at 1076 cm^{-1} caused a major decrease in accuracy compared to the rest of wavenumber values.

If we compared the two approaches:

```
# comparing the two approaches
```

```
fig, (ax, ax2) = plt.subplots(2,1, tight_layout=True)

feature_importance.plot.bar(x='Wavenumbers', y='Coefficients', ax=ax, color='r')
forest_importances.plot.bar(yerr=r.importances_std, ax=ax2, color='g')
```

```
<Axes: >
```



Some wavenumbers matches, some other do not.

3.1.6 Permutation importance using RF

Let's do the same process but with a RF model. After we created the pipeline and and test it on the test set. We got an accuracy of 0.9. We access the feature importance using the attribute `feature_importances_`

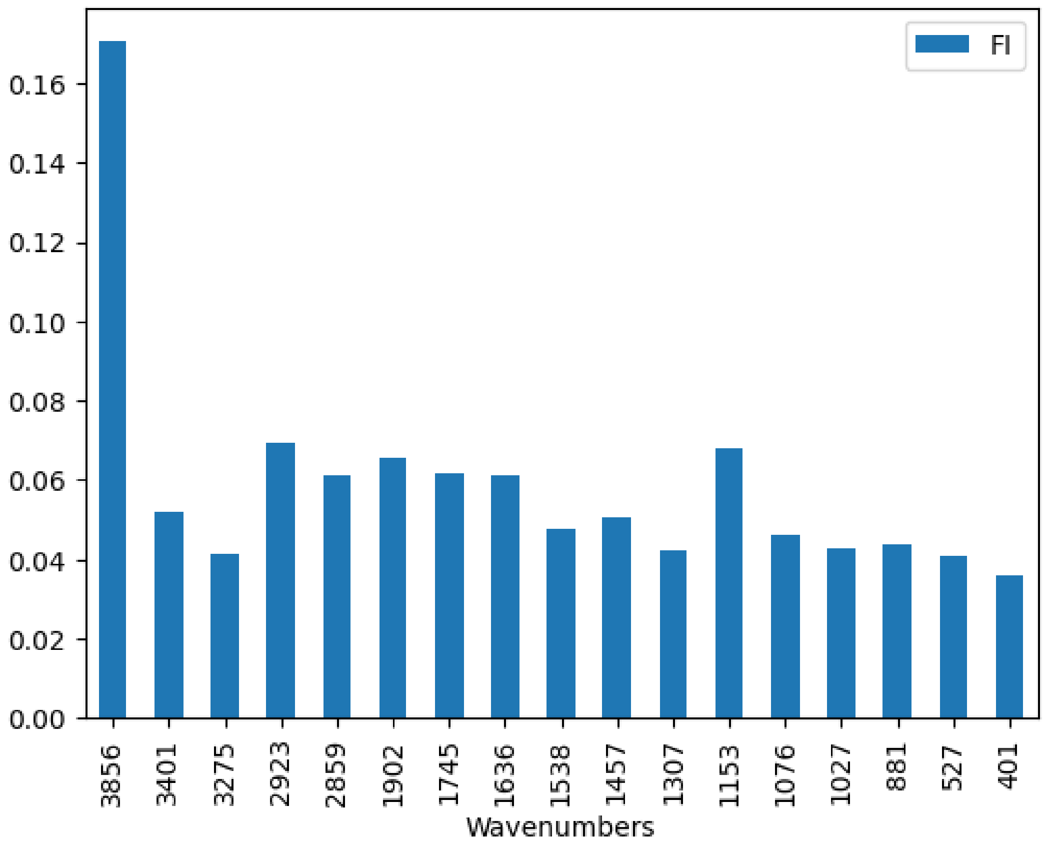
```
pipe.named_steps['model'].feature_importances_
```

```
array([0.17048121, 0.05187198, 0.04152147, 0.06929754, 0.06114025,
       0.06534715, 0.06150834, 0.0610971 , 0.04741274, 0.05061926,
       0.04238396, 0.06812852, 0.04599945, 0.04263272, 0.04370885,
       0.040633  , 0.03621647])
```

We can plot the values vs wavenumber values

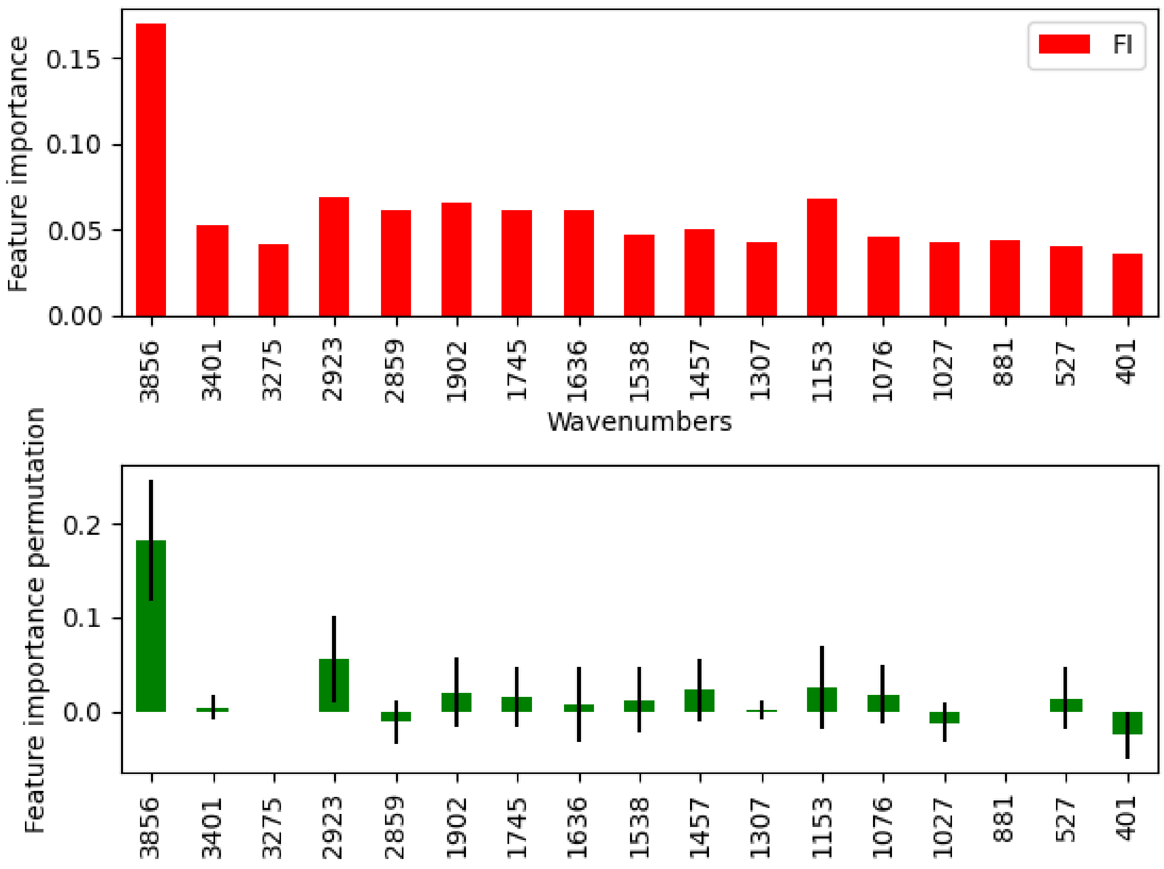
```
feature_importance = pd.DataFrame()
feature_importance['Wavenumbers'] = wavenumbers
feature_importance['FI'] = pipe.named_steps['model'].feature_importances_
feature_importance['Wavenumbers'] = feature_importance['Wavenumbers'].astype('category')
feature_importance.plot.bar(x='Wavenumbers', y='FI')
```

```
<Axes: xlabel='Wavenumbers'>
```



The absorbance at 3856 cm^{-1} seem to be quite important for this model. If we compare to the permutation imporance we can see that they are quite similar.

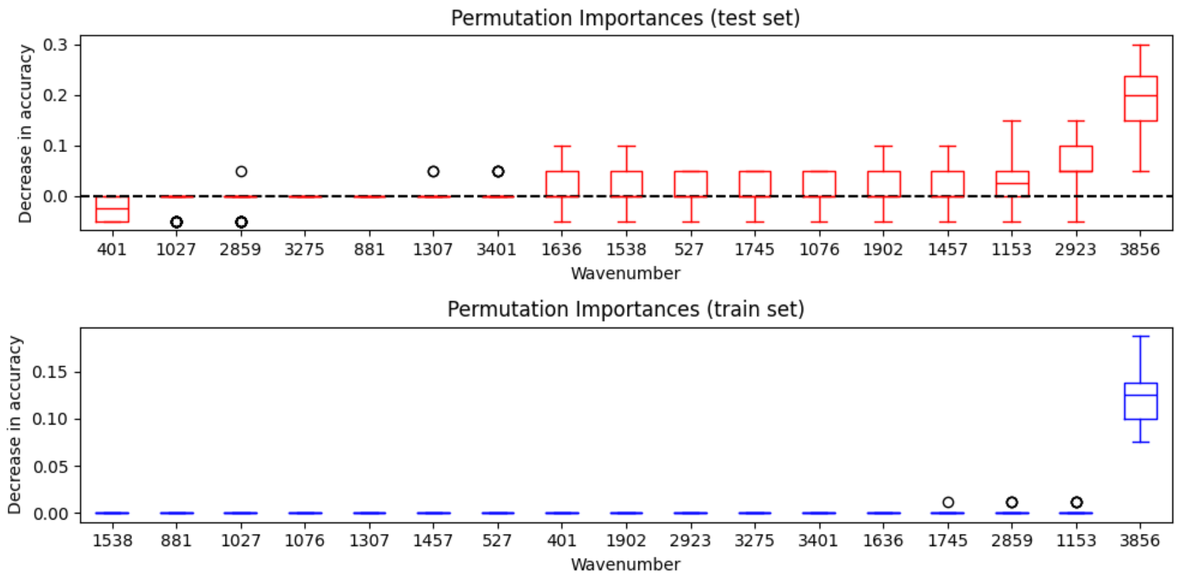
```
Text(0, 0.5, 'Feature importance permutation')
```



You can also compare between using the train set or the test set for permutatioin importance:

```
fig, (ax, ax2) = plt.subplots(2,1,figsize=(10,5), tight_layout=True)
importances.plot.box(vert=True, whis=10, ax=ax, color='r')
ax.axhline(0, ls='--', color='k')

importances_train.plot.box(vert=True, whis=10, ax=ax2, color='b')
ax.axhline(0, ls='--', color='k')


ax.set_xlabel("Wavenumber")
ax.set_ylabel("Decrease in accuracy")
ax.set_title("Permutation Importances (test set)")

ax2.set_xlabel("Wavenumber")
ax2.set_ylabel("Decrease in accuracy")
ax2.set_title("Permutation Importances (train set)")
```

```
Text(0.5, 1.0, 'Permutation Importances (train set)')
```



Something to consider and I quote:

"Permutation importances can be computed either on the training set or on a held-out testing or validation set. Using a held-out set makes it possible to highlight which features contribute the most to the generalization power of the inspected model. Features that are important on the training set but not on the held-out set might cause the model to overfit."

Warning

Features that are deemed of low importance for a bad model (low cross-validation score) could be very important for a good model. Therefore it is always important to evaluate the predictive power of a model using a held-out set (or better with cross-validation) prior to computing importances. Permutation importance does not reflect to the intrinsic predictive value of a feature by itself but how important this feature is for a particular model (sklearn documentation)

3.2 Nested Cross validation

Nested cross validation can be useful when you do not have a large sample size. Improper use of validation approaches can result in biases in our results. Nested CV and train/test split approaches produce robust and unbiased performance estimates regardless of sample size. Grafical examples of how different validations techniques are shown in Figure 3.1. Nested cross-validation has a inner and outer layer. The inner layer will be used to optimize the model and the outer layer will be used to test the optimized model. This process will be repeated n times. Bear in mind that this is a very time and source consuming approach. You can read more on which different approaches you can use to deal with small data sets [Vabalas et al. (2019)](Varma and Simon 2006). For this example we are going to use nested cross validation as shown in Figure 3.1. We take all the data, and we split it in n folds (outer layer), n-1 folds will be used for model development/optimization (inner layer) and the reminder fold will be used to validate the model. This process will be repeated for each fold in the outer layer. The final accuracy will be the mean of each outer layer.

Validation methods. A: Train/Test Split. B: K-Fold CV. C: Nested CV. D: Partially nested CV. ACC—overall accuracy of the model, ACCi.—accuracy in a single CV fold (Taken from (Vabalas et al. 2019))

As we assumed that we do not have enough samples in our data set, therefore, we do not need to split the data into train and test sets.

```
# Split features and target
X = df.loc[:,"4000":"403"]
y = df.loc[:,"Exposed"]
```

We set up the conditions of our nested cross validations. We add our model and any preprocessing, the number of splits for the inner and outer layer and our final pipline. This time, we are using KFold and 3 folds for both layers

```
scaler = StandardScaler()
model = LogisticRegression(max_iter=10000)
splits_outer = 3
splits_inner = 3

pipe = Pipeline(steps=[("scaler", scaler), ("model", model)])

# configure nested cross-validation layers
cv_outer = KFold(n_splits=splits_outer, shuffle=True, random_state=123)
cv_inner = KFold(n_splits=splits_inner, shuffle=True, random_state=123)
```

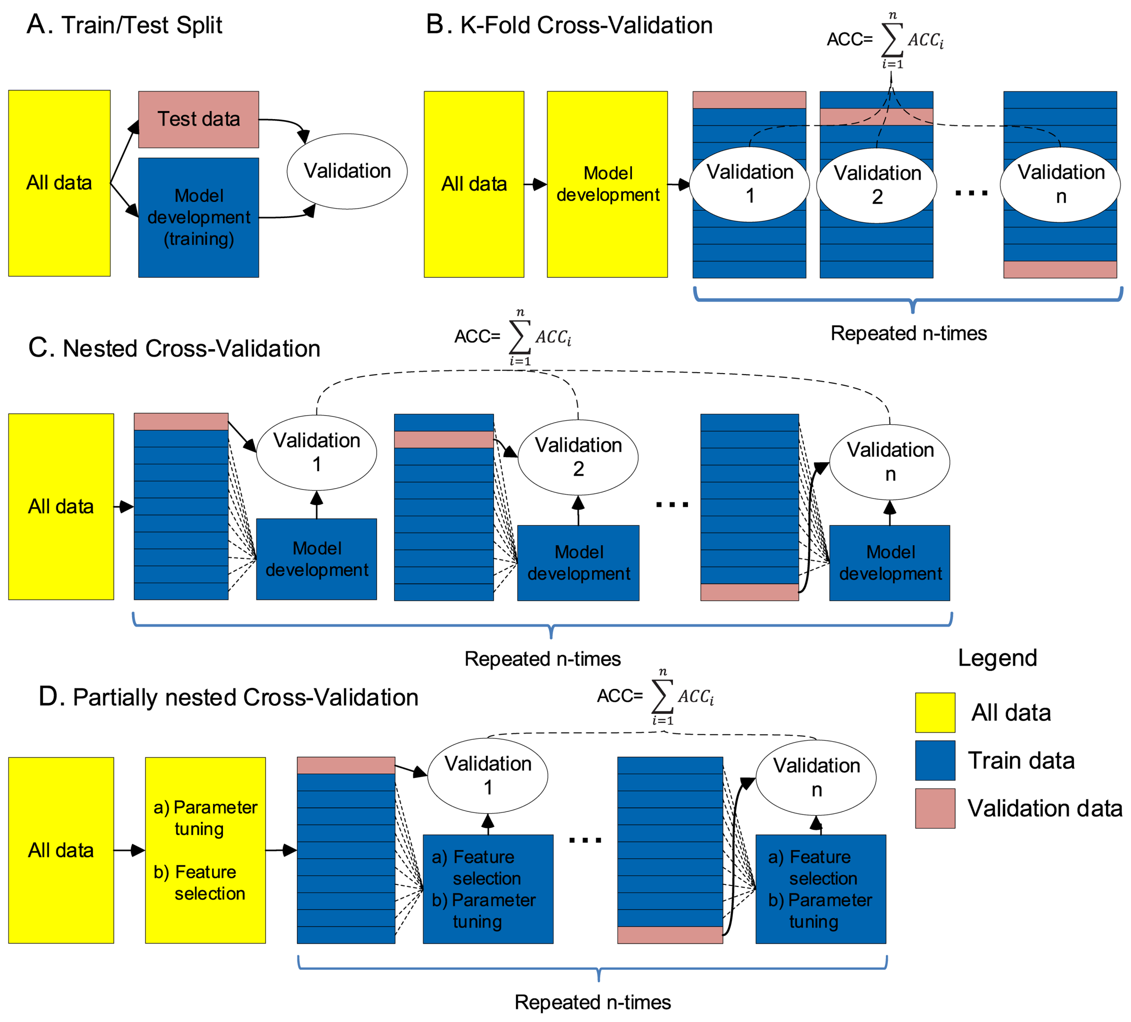


Figure 3.1: validation-techniques

Now, the for loop. First, we split the data using the outer split,with `for train_ix, test_ix in cv_outer.split(X):`. We run GridSearchCV using the train test with the inner layer cross validation (specified in the parameter cv of GridSearchCV). The best modelis then tested on the test set of the outer layer.

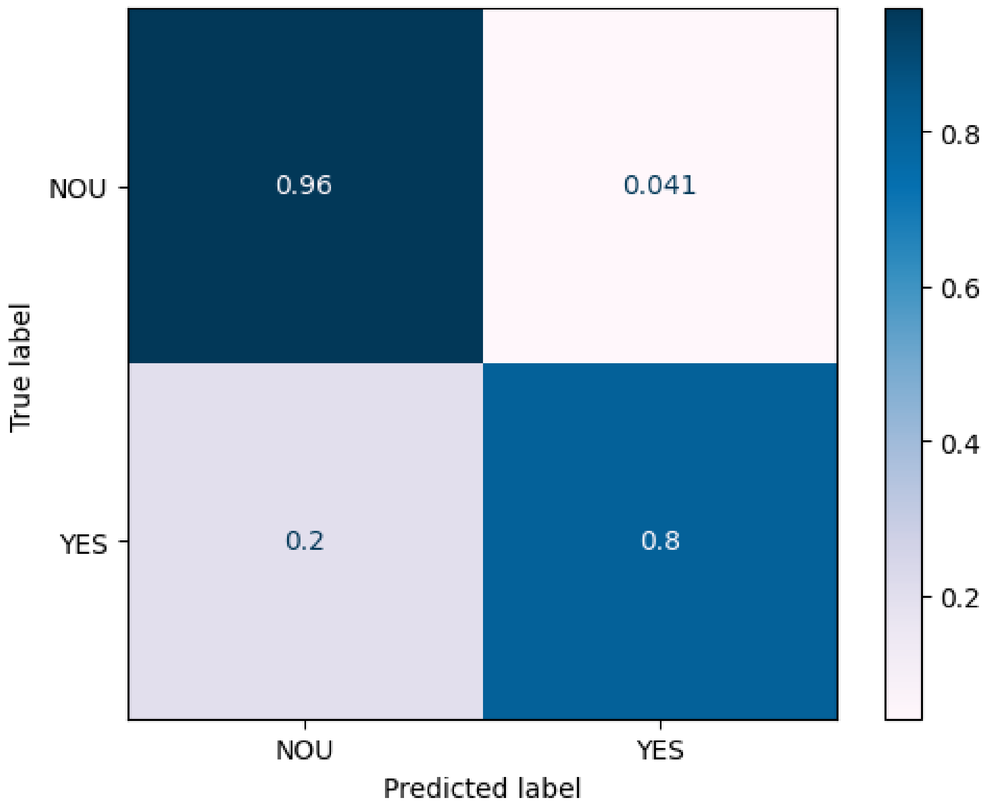
```
# create confusion matrix list to save each of external cv layer
cm_nested = []

# enumerate splits and create AUC plot
yhat_nested = []
y_test_nested = []
X_test_nested = []
best_estimators = []
mean_fpr = np.linspace(0, 1, 100)
outer_results = list()


for train_ix, test_ix in cv_outer.split(X):
# split data
    X_train, X_test = X.iloc[train_ix, :], X.iloc[test_ix, :]
    y_train, y_test = y[train_ix], y[test_ix]

    # define search space
    param_grid = {'model__penalty': ['l1', 'l2'],
                  'model__C': [0.01, 0.1, 1, 10],
                  'model__solver': ['liblinear', 'saga']
                  }
    # define search

    search = GridSearchCV(pipe, param_grid, scoring='accuracy', cv=cv_inner, refit=True)
    # execute search
    result = search.fit(X_train, y_train)

    # get the best performing model fit on the whole training set
    best_model = result.best_estimator_

    # create pipeline with the best model
    best_pipe = Pipeline(steps=[("scaler", scaler), ("best_model", best_model)])
    best_pipe.fit(X_train, y_train)
    # evaluate model on the hold out dataset

    yhat = best_pipe.predict(X_test)
    # evaluate the model
    acc = accuracy_score(y_test, yhat)
    cm = confusion_matrix(y_test, yhat)
    yhat_nested.append(yhat)

    y_test_nested.append(y_test)
    X_test_nested.append(X_test)

     # store the result
    outer_results.append(acc)
    cm_nested.append(cm)
    best_estimators.append(best_pipe)

    print(f'Accuracy = {result.best_score_:.2f}, Parameters={result.best_params_}')

# summarize the estimated performance of the model
print(f'Final accuracy = {np.mean(outer_results):.2f} ±  {np.std(outer_results):.2f}')
```

```
Accuracy = 0.92, Parameters={'model__C': 10, 'model__penalty': 'l1', 'model__solver': 'libli
Accuracy = 0.87, Parameters={'model__C': 10, 'model__penalty': 'l2', 'model__solver': 'libli
Accuracy = 0.85, Parameters={'model__C': 10, 'model__penalty': 'l1', 'model__solver': 'libli
Final accuracy = 0.88 ±  0.05
```

To report the result you can use AUC-ROC with confidence intervals or a confusion matrix with the mean of the accuracies of each of the results from the outer layer. Let's plot the confusion matrix

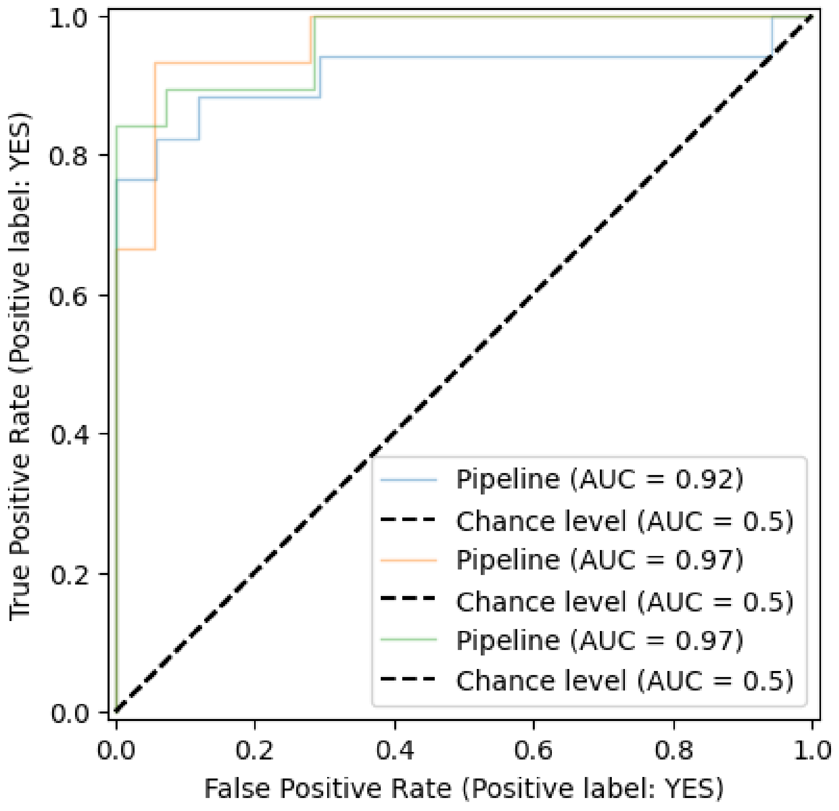
```
# sum all the confusion matrices
cm_sum = np.sum(cm_nested,axis=0)

# we normalised the confusion matrix
cm_sum_normalized = cm_sum.astype('float')/cm_sum.sum(axis=1)[:, np.newaxis]

# Plot the confusion matrix
disp = ConfusionMatrixDisplay(confusion_matrix=cm_sum_normalized, display_labels=best_mode
disp.plot(cmap = "PuBu")
```

```
<sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at 0x15e922240>
```



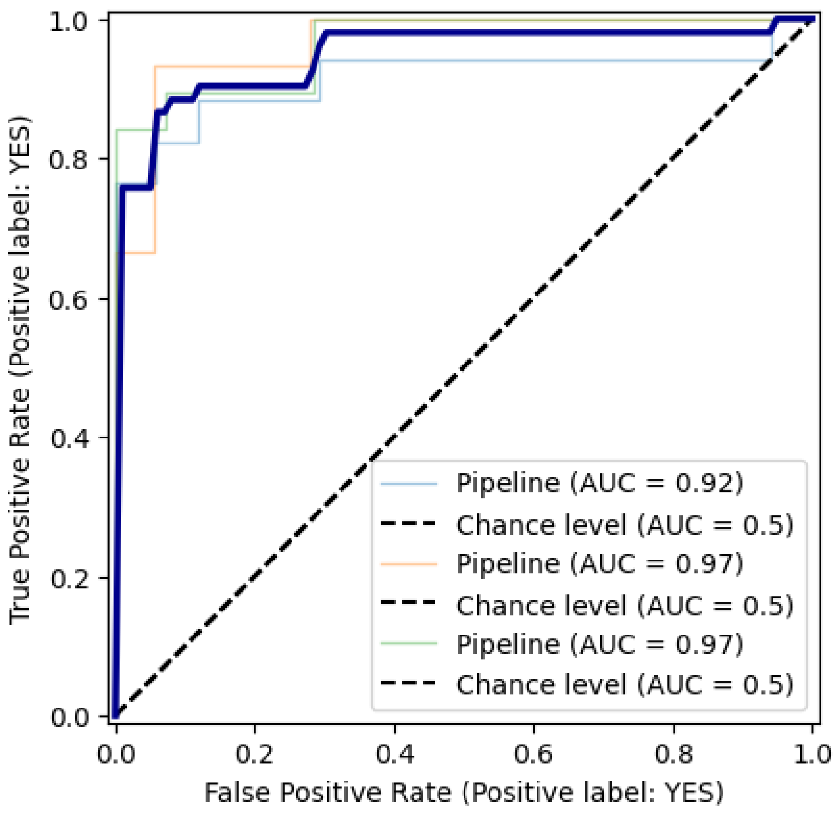
To plot the AUC-ROC with confidence intervals is a little bit tricky using nested cross validation. We need to create lists where we store the best estimator and the X_train, and y_train sets of each outer layer. Then we run a for loop where we going to plot the roc curve for each X_train, y_test and best model using `RocCurveDisplay.from_estimator` function.

```
# AUC-ROC curve

mean_fpr = np.linspace(0, 1, 100)
tprs = []
aucs = []


fig, ax = plt.subplots()
for a, b, c in zip(X_test_nested,y_test_nested, best_estimators):
        viz = RocCurveDisplay.from_estimator(
        c,
        a,
        b,
        alpha=0.4,
        lw=1,
        plot_chance_level = True,
        ax=ax)

    interp_tpr = np.interp(mean_fpr, viz.fpr, viz.tpr)
    interp_tpr[0] = 0.0
    tprs.append(interp_tpr)
    aucs.append(viz.roc_auc)
```



We can see three curves that correspond to each outer layer and we can see the values of AUC for each one in the plot legend. Now we need to calculate the mean of the curves plot it over the current plot

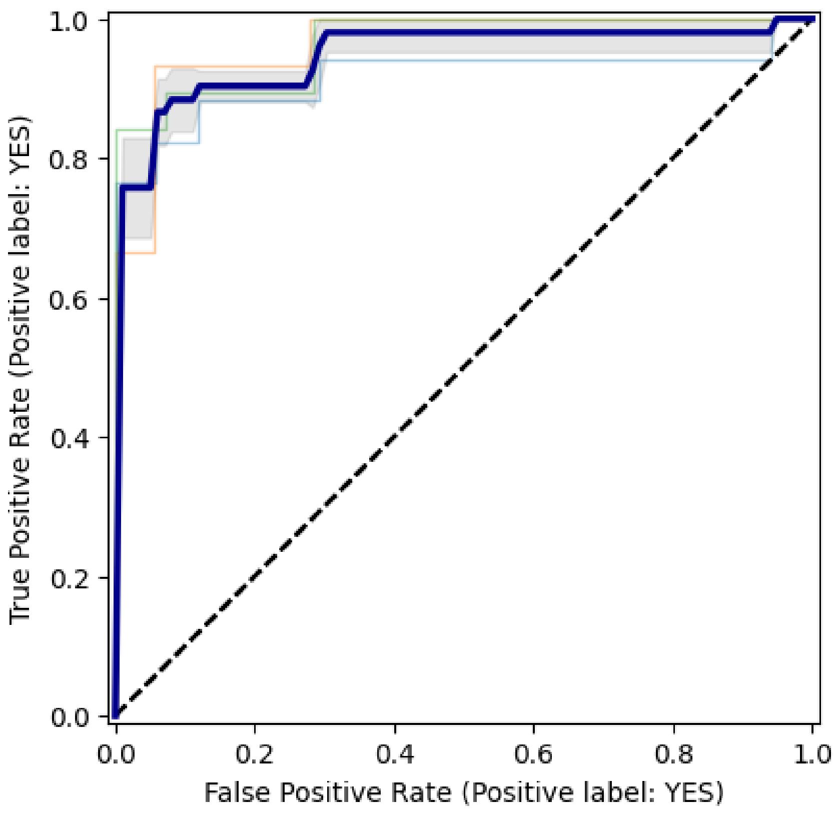
```
mean_tpr = np.mean(tprs, axis=0)
mean_tpr[-1] = 1.0
mean_auc = auc(mean_fpr, mean_tpr)
std_auc = np.std(aucs)

fig, ax = plt.subplots()
for a, b, c in zip(X_test_nested,y_test_nested, best_estimators):
        viz = RocCurveDisplay.from_estimator(
        c,
        a,
        b,
        alpha=0.4,
        lw=1,
        plot_chance_level = True,
        ax=ax)

        interp_tpr = np.interp(mean_fpr, viz.fpr, viz.tpr)
        interp_tpr[0] = 0.0
        tprs.append(interp_tpr)
        aucs.append(viz.roc_auc)

ax.plot(
        mean_fpr,
        mean_tpr,
        color="darkblue",
        label=r"Mean ROC (AUC = %0.2f $\pm$ %0.2f)" % (mean_auc, std_auc),
        lw=2.5,
        alpha=1)
```



Finally, we plot the standard deviation and get rid of the legend

```
fig, ax = plt.subplots()
for a, b, c in zip(X_test_nested,y_test_nested, best_estimators):
        viz = RocCurveDisplay.from_estimator(
        c,
        a,
        b,
        alpha=0.4,
        lw=1,
        plot_chance_level = True,
        ax=ax)

        interp_tpr = np.interp(mean_fpr, viz.fpr, viz.tpr)
        interp_tpr[0] = 0.0
        tprs.append(interp_tpr)
        aucs.append(viz.roc_auc)
```

```
ax.plot(
        mean_fpr,
        mean_tpr,
        color="darkblue",
        label=r"Mean ROC (AUC = %0.2f $\pm$ %0.2f)" % (mean_auc, std_auc),
        lw=2.5,
        alpha=1)

std_tpr = np.std(tprs, axis=0)
tprs_upper = np.minimum(mean_tpr + std_tpr, 1)
tprs_lower = np.maximum(mean_tpr - std_tpr, 0)
ax.fill_between(
        mean_fpr,
        tprs_lower,
        tprs_upper,
        color="grey",
        alpha=0.2)

ax.get_legend().remove()
```



3.3 Nested cross validation plus validation on unseen data

This is a variation of the nested cross validation (develop by our colleagues from IHI). The addition is that you will test the final model in unseen data. So, you can report both, your results of the nested cross-validation as well as the accuracy on the unseen data.

Here we are going to use:

1. Using numpy arrays instead of dataframes
2. Split the data into train and test as a whole
3. Saving a lot of data from the cross validation to further explore the results.
4. Export and save the model (to avoid training every time we run the notebook)
5. Setting up the general aesthetics of the plots at the beginning of the script

For easiness, we have contain the whole code into a function, to avoid mistakes when copy and paste from different notebooks

3.3.1 Set plot aesthetics

Setting up the general aesthetics of the plot. When using seaborn packgaes, you can set the syle, context fonts, of all the plots from the beginning

```
sns.set(
    context = "paper",
    style = "white",
    palette = "deep",
    font_scale = 2.0,
    color_codes = True,
    rc = ({"font.family": "Dejavu Sans"})
    )
```

3.3.2 Defining the fuctions

We defined two functions: ml_loop and confusion_matrix_mau.

```
# define functions

def ml_loop(
    X_train,
    y_train_species,
    kf,
    num_rounds: int,
    random_grid: dict,
    scoring: str,
    species_model,
    name
):
    """
    Perform a machine learning loop with cross-validation and randomized grid search.

    This function performs a machine learning loop that includes cross-validation,
    randomized grid search for hyperparameter tuning, and model evaluation. It
    trains and evaluates the model multiple times (specified by num_rounds) and
    collects the results.

    Parameters
    ----------
    X_train : array-like of shape (n_samples, n_features)
```

```
        The training input samples.
    y_train_species : array-like of shape (n_samples,)
        The true labels for the training samples.
    kf : cross-validation generator
        A cross-validation generator, such as KFold or StratifiedKFold.
    num_rounds : int
        The number of rounds to run the loop.
    random_grid : dict
        The hyperparameter grid for randomized search.
    scoring : str
        The scoring metric to evaluate the model, e.g., 'accuracy'.
    species_model : estimator object
        The machine learning model to be used, e.g., an instance of SVC.

    Returns
    -------
    species_predicted : list
        A list of predicted labels for the test samples across all rounds.
    species_true : list
        A list of true labels for the test samples across all rounds.
    kf_results : pd.DataFrame
        A DataFrame containing the model parameters and global accuracy scores.
    kf_per_class_results : list
        A list of per-class accuracy scores for


    """
    start = time()
    kf_results = pd.DataFrame() # model parameters and global accuracy score
    kf_per_class_results = []
    species_predicted, species_true = [], []

    for vuelta in range(num_rounds):
        print(f"\nThis is round {vuelta+1}...\n")
        SEED = np.random.randint(0, 81478)

        # Before your cross-validation loop
        # y_species_train = np.where(y_species_train == 'AA', 0, 1)

        # Inside your cross-validation loop, this step is not needed as the data is alread
```

```
# cross validation and splitting of the validation set
for train_index, test_index in kf.split(X_train, y_train_species):
    X_train_set, X_test_set = X_train[train_index], X_train[test_index]
    y_train_species_set, y_val_species_set = (
        y_train_species[train_index],
        y_train_species[test_index],
    )

    # print('The shape of X train set : {}'.format(X_train_set.shape))
    # print('The shape of y train species : {}'.format(y_train_species_set.shape))
    # print('The shape of X test set : {}'.format(X_test_set.shape))
    # print('The shape of y test species : {}\n'.format(y_val_species_set.shape))

    # generate models using all combinations of settings

    # RANDOMSED GRID SEARCH
    # Random search of parameters, using 5 fold cross validation,
    # search across 100 different combinations, and use all available cores

    n_iter_search = 10
    rsCV = RandomizedSearchCV(
        verbose=1,
        estimator=species_model,
        param_distributions=random_grid,
        n_iter=n_iter_search,
        scoring=scoring,
        cv=kf,
        refit=True,
        n_jobs=-1,
    )

    rsCV_result = rsCV.fit(X_train_set, y_train_species_set)

    # print out results and give hyperparameter settings for best one
    means = rsCV_result.cv_results_["mean_test_score"]
    stds = rsCV_result.cv_results_["std_test_score"]
    params = rsCV_result.cv_results_["params"]
    # for mean, stdev, param in zip(means, stds, params):
    # print("Accuracy of %.2f $\pm$(%.2f) with: %r" % (mean, stdev, param))

    # print best parameter settings
```

```
        print(
            "Best accuracy: %.2f using %s"
            % (rsCV_result.best_score_, rsCV_result.best_params_)
        )

        # Insert the best parameters identified by randomized grid search into the bas
        species_classifier = species_model.set_params(**rsCV_result.best_params_)

        # Fit your models
        species_classifier.fit(X_train_set, y_train_species_set)

        # predict test instances
        sp_predictions = species_classifier.predict(X_test_set)

        # zip all predictions for plotting averaged confusion matrix
        # species
        for predicted_sp, true_sp in zip(sp_predictions, y_val_species_set):
            species_predicted.append(predicted_sp)
            species_true.append(true_sp)

        # species local confusion matrix & classification report
        local_cm_species = confusion_matrix(y_val_species_set, sp_predictions)
        local_report_species = classification_report(
            y_val_species_set, sp_predictions
        )

        # append feauture importances
        # local_feat_impces_species = pd.DataFrame(species_classifier.feature_importan
        #                                    index = features.columns).sort_values(by

        # summarizing results
        local_kf_results = pd.DataFrame(
            [
                (
                    "Accuracy_species",
                    accuracy_score(y_val_species_set, sp_predictions),
                ),
                ("TRAIN", str(train_index)),
                ("TEST", str(test_index)),
                ("CM", local_cm_species),
                ("Classification report", local_report_species),
                    ("y_test", y_val_species_set),
                    # ("Feature importances", #local_feat_impces_species.to_dict())
                ]
            ).T

            local_kf_results.columns = local_kf_results.iloc[0]
            local_kf_results = local_kf_results[1:]
            kf_results = pd.concat(
                [kf_results, local_kf_results], axis=0, join="outer"
            ).reset_index(drop=True)

            # per class accuracy
            local_support = precision_recall_fscore_support(
                y_val_species_set, sp_predictions
            )[3]
            local_acc = np.diag(local_cm_species) / local_support
            kf_per_class_results.append(local_acc)
    elapsed = time() - start
    print(f"\nTime elapsed: {elapsed:.2f} seconds")

    filename = "./results/models/trained_model_" + name + ".sav"
    joblib.dump(species_classifier, filename)

    return species_true, species_predicted, kf_results

def confusion_matrix_mau(y_pred, y_true, xticks_rotation=None, ax=None):
    """
    Displays a normalized confusion matrix using the true and predicted labels.

    This function generates and displays a normalized confusion matrix using the
    true labels (`y_true`) and the predicted labels (`y_pred`). The confusion
    matrix is displayed using a grayscale colormap and is normalized to show
    proportions instead of raw counts.

    Args:
        y_pred (array-like of shape (n_samples,)): The predicted labels.
        y_true (array-like of shape (n_samples,)): The true labels.
        ax (matplotlib.axes.Axes, optional): The axes on which to plot the confusion matri
            If None, the current axes will be used.

    Returns:
        None
    """
    ConfusionMatrixDisplay.from_predictions(y_pred=y_pred, y_true=y_true,normalize='true',
```

The only new package here is `joblib` to save our trained model. The principle is the same as nested cross validation.

3.3.3 Split the data

We load our data and split into train and test set. We also stratify the split using the "Exposed" variable and then separate our features and target from each of the splits

```
X_train = np.asarray(train_set.iloc[:,5:-1]) # feature matrix
y_train_exposure = np.asarray(train_set['Exposed'])

X_test = np.asarray(test_set.iloc[:,5:-1])
y_test_exposure = np.asarray(test_set['Exposed'])
```

Why are we using numpy arrays instead of dataframes? Numpy arrays provide a consistent and standardized data format that scikit-learn classifiers can easily handle. This allows for integration with other scikit-learn functionalities, such as preprocessing, cross-validation, and model evaluation

Tip

Sklearn classifiers can easyly handle numpy arrays

3.3.4 Setting the parameters of nested cross validation

Here, we defined the number of folds (inner layer) and the number of rounds (outer layer). Note that we are using a random number generator as random_seed. This number will change each round and will give use a different split in the outer layer

```
# Set validation procedure
num_folds = 5 # split training set into 5 parts for validation
num_rounds = 2 # increase this to 5 or 10 once code is bug-free
seed = 42 # pick any integer. This ensures reproducibility of the tests
random_seed = np.random.randint(0, 81478)

scoring = 'accuracy' # score model accuracy
```

```
# cross validation strategy
kf = KFold(
        n_splits = num_folds,
        shuffle = True,
        random_state = random_seed
        )
```

3.3.5 Setting the parameters of nested cross validation

The hyperparameters for support vector machine that we are going to test are C, kernel and gamma.

```
random_grid = {'C': [10, 1.0, 0.1, 0.01], 'kernel': ["linear", 'rbf', 'poly'], 'gamma': [0

species_model = SVC()
```

3.3.6 Run the loop

We run the loop. The function needs X and y train, the split type, number of rounds and folds, hyperparameters, the score and the model

```
# Species prediction

y_mau_true, y_mau_predicted, main_results = ml_loop(X_train, y_train_exposure, kf, num_rou
```

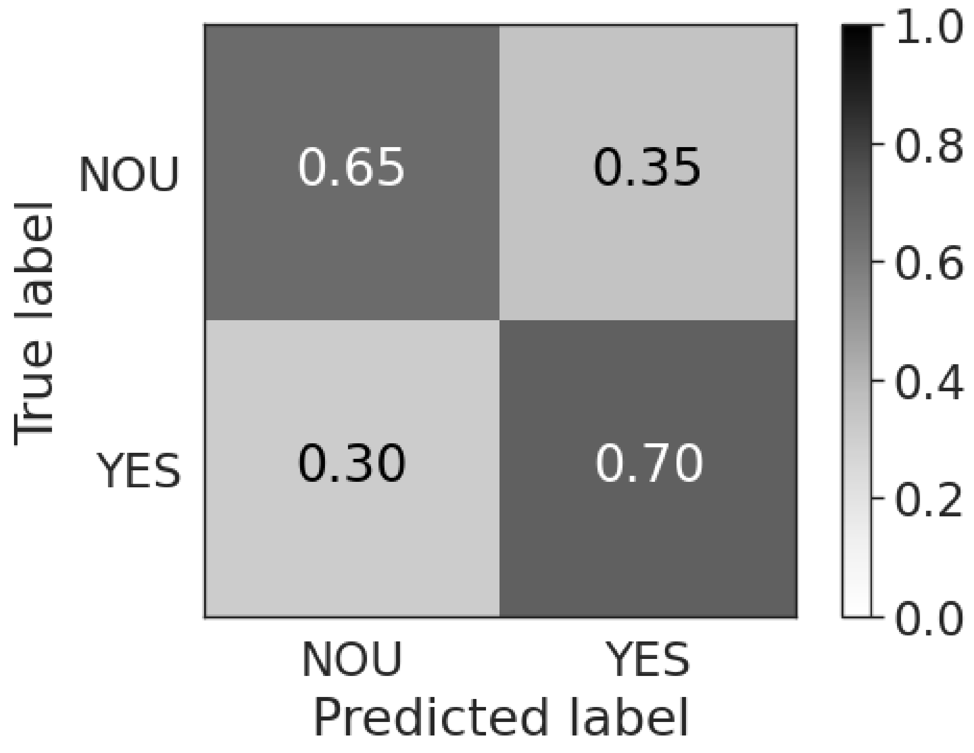
```
This is round 1...

Fitting 5 folds for each of 10 candidates, totalling 50 fits
Best accuracy: 0.78 using {'kernel': 'poly', 'gamma': 10, 'C': 10}
Fitting 5 folds for each of 10 candidates, totalling 50 fits
Best accuracy: 0.75 using {'kernel': 'poly', 'gamma': 10, 'C': 1.0}
Fitting 5 folds for each of 10 candidates, totalling 50 fits
Best accuracy: 0.86 using {'kernel': 'poly', 'gamma': 1, 'C': 10}
Fitting 5 folds for each of 10 candidates, totalling 50 fits
Best accuracy: 0.77 using {'kernel': 'poly', 'gamma': 10, 'C': 0.1}
Fitting 5 folds for each of 10 candidates, totalling 50 fits
Best accuracy: 0.74 using {'kernel': 'poly', 'gamma': 1, 'C': 1.0}
```

```
This is round 2...

Fitting 5 folds for each of 10 candidates, totalling 50 fits
Best accuracy: 0.78 using {'kernel': 'poly', 'gamma': 10, 'C': 1.0}
Fitting 5 folds for each of 10 candidates, totalling 50 fits
Best accuracy: 0.75 using {'kernel': 'poly', 'gamma': 1, 'C': 10}
Fitting 5 folds for each of 10 candidates, totalling 50 fits
Best accuracy: 0.86 using {'kernel': 'poly', 'gamma': 10, 'C': 0.01}
Fitting 5 folds for each of 10 candidates, totalling 50 fits
Best accuracy: 0.77 using {'kernel': 'poly', 'gamma': 10, 'C': 10}
Fitting 5 folds for each of 10 candidates, totalling 50 fits
Best accuracy: 0.74 using {'kernel': 'poly', 'gamma': 1, 'C': 1.0}

Time elapsed: 3.40 seconds
```

You can see how each round will contain 5 best models.

We plot a confusion matrix which is the sum of all the confusion matrices from each round.

```
# this function make pretty confusion matrices
confusion_matrix_mau(y_true=y_mau_true, y_pred=y_mau_predicted)
```

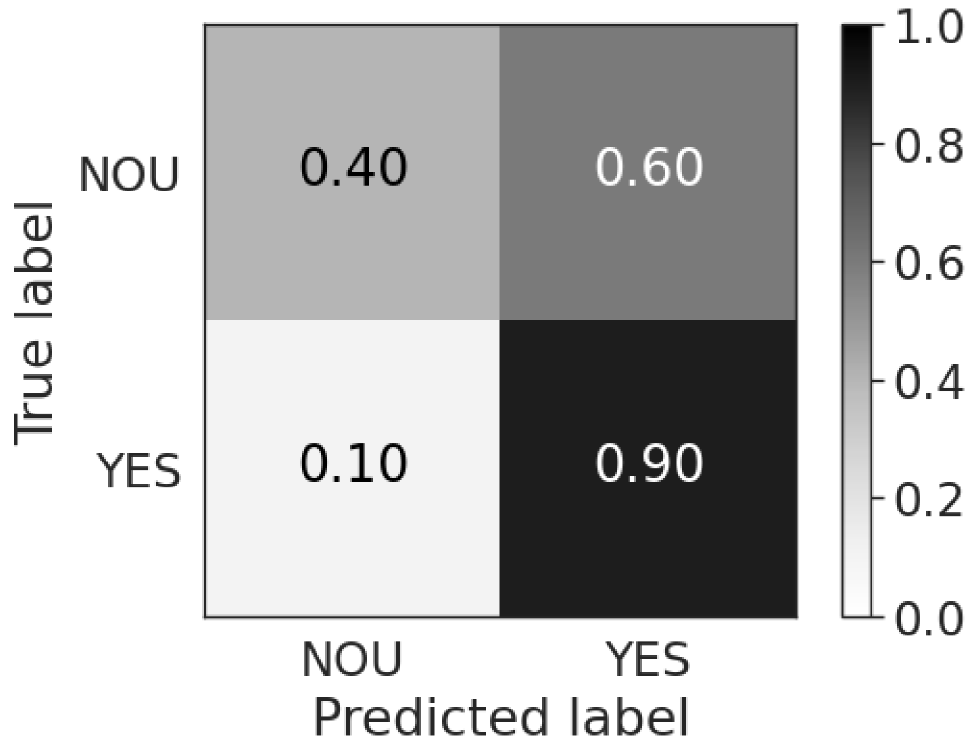


We load our trained model, test it in the unseen data and plot the confusion matrix

```
loaded_model_exposure = joblib.load('./results/models/trained_model_mau_mau.sav')
```

```
y_hd_pred = loaded_model_exposure.predict(X_test)
acc = accuracy_score(y_true=y_test_species, y_pred=y_hd_pred)
print(f"Accuracy on test set using the head: {acc}")

confusion_matrix_mau(y_true=y_test_species, y_pred=y_hd_pred)
```

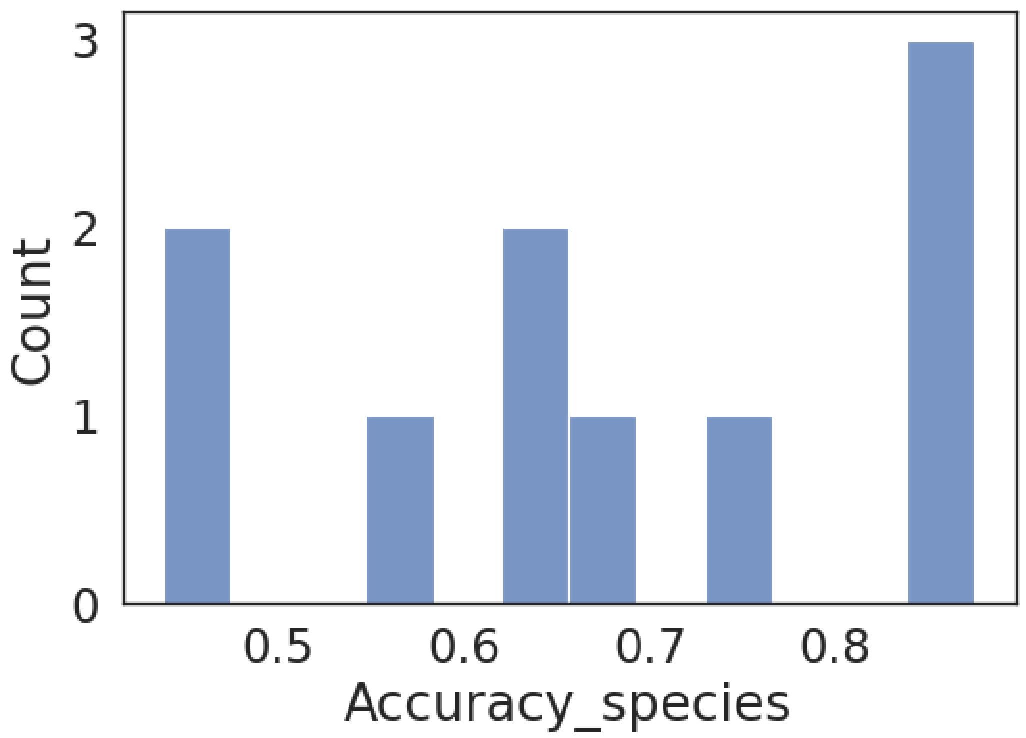
```
Accuracy on test set using the head: 0.65
```



Since we save a lot of data from the loop, for example accuracy of each inner layer. We can plot an histogram to check how stable was the accuracy across layers.

```
sns.histplot(data=main_results, x='Accuracy_species',bins=12)
```

```
<Axes: xlabel='Accuracy_species', ylabel='Count'>
```


3
2
1
0
Count
0.5
0.6
0.7
0.8
Accuracy_species

4 Deep Learning pipelines

Deep learning is a subarea of machine learning that focuses on the use of Artifical Neural Networks (ANN). These type of models have shown to be quite powerfull for infrared spectroscopy. For deep learning analysis, we are going to use TensorFlow. For instructions to install you visit [https://keras.io/]. As always, read the documentation. They have a lot of really well put tutorials to get you started.

4.1 A basic artificial neural network for species classification

The most basic artifical neural network is the Multilayer Perceptron (MLP). It has following architecture (Figure 4.1):

- One input layer
- One or more hidden layers
- One final layer called the output layer.

Every layer except the output layer includes a bias neuron and is fully connected to the next layer.

We will start with a very simple ANN to predict two species of Anopheles mosquitoes using infrared spectroscopy.

4.1.1 Load and preprocess the data

As mentioned early, we are going to use Tensorflow and Keras to do our deep learning analsysis. Let's import the packages.

```
import tensorflow as tf
from tensorflow import keras
from keras import regularizers
import tensorflow_docs as tfdocs
import tensorflow_docs.modeling
```

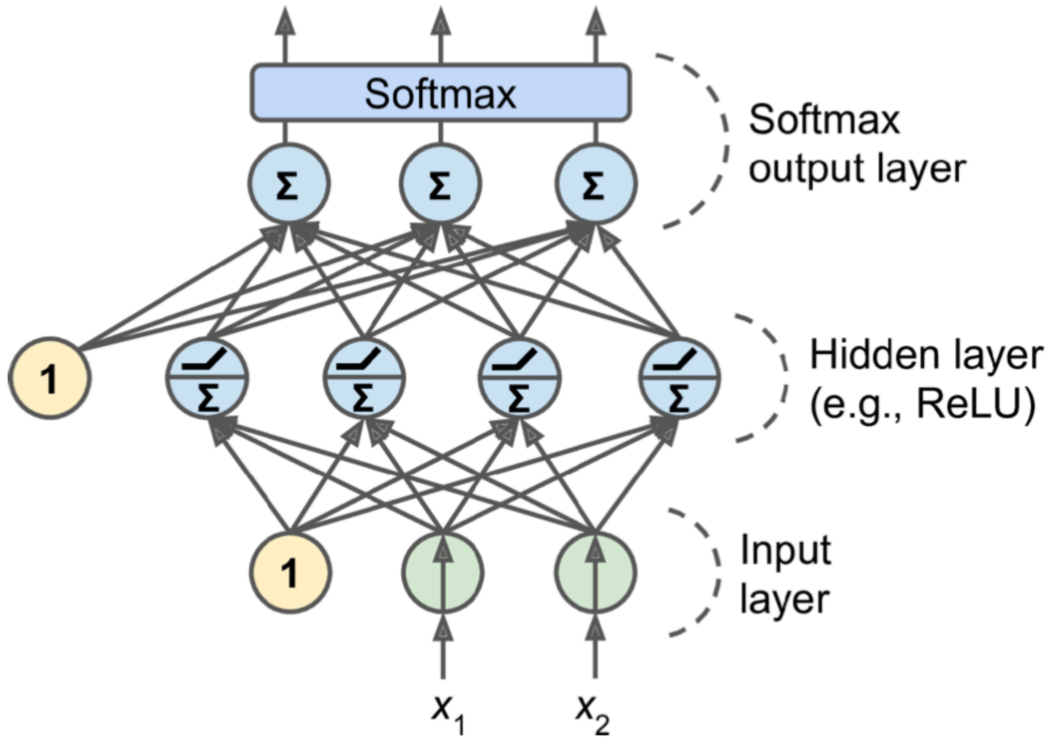


Figure 4.1: mpl_architecture

```python
import pandas as pd
import numpy as np

import matplotlib.pyplot as plt
from sklearn.preprocessing import LabelBinarizer
from sklearn.decomposition import PCA
from sklearn.model_selection import train_test_split
```

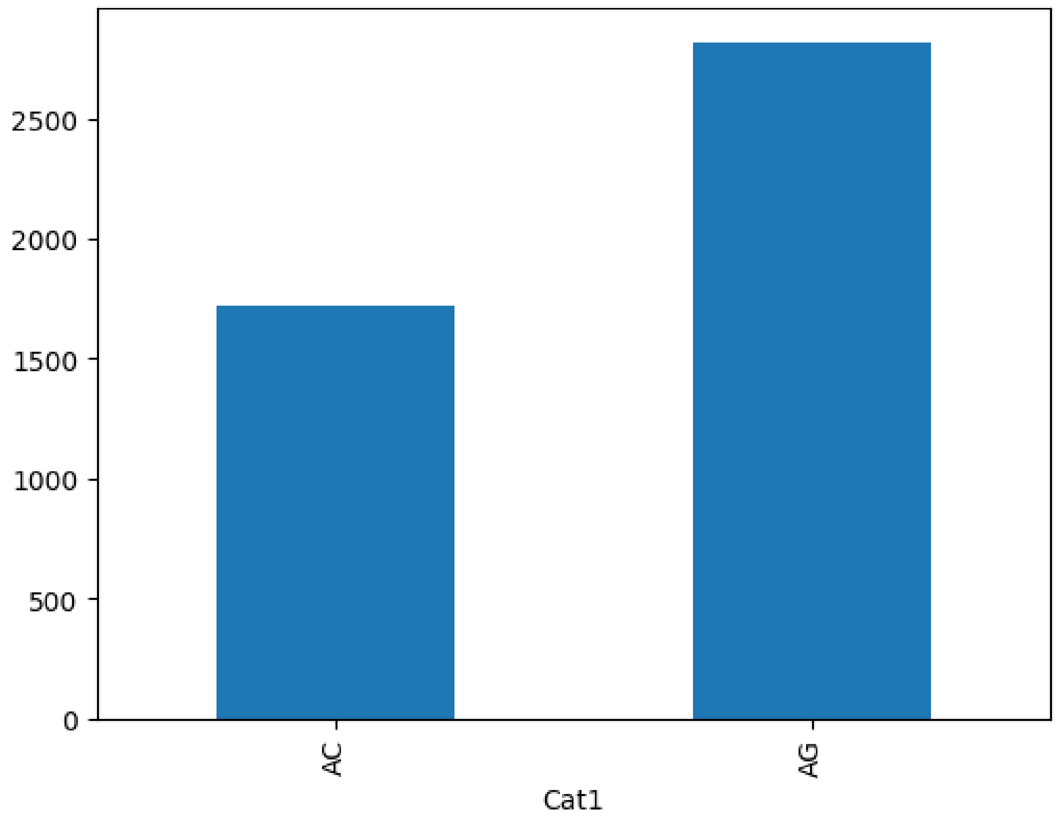
Our dataset contains more than 10 categorical targets and around 1812 features.

	Cat1	Cat2	Cat3	Cat4	Cat5	Cat6	Cat7	Cat8	Cat9	Cat10	...	420	418	416
0	AC	S	0	YY	SU	T1	C1	R1	190823	111223	...	0.2321	0.2309	0.22
1	AC	S	0	YY	SU	T1	C1	R1	190823	111223	...	0.2223	0.2198	0.21
2	AC	S	0	YY	SU	T1	C1	R1	190823	111223	...	0.2040	0.2028	0.20
3	AC	S	0	YY	SU	T1	C1	R1	190823	111223	...	0.2409	0.2389	0.23
4	AC	S	0	YY	SU	T1	C1	R1	190823	111223	...	0.2150	0.2155	0.21
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
4539	AG	S	31	YY	SU	T1	K2	R4	190923	30124	...	0.1638	0.1634	0.16
4540	AG	S	31	YY	SU	T1	K2	R4	190923	30124	...	0.2457	0.2438	0.24
4541	AG	S	31	YY	SU	T1	K2	R4	190923	30124	...	0.2676	0.2667	0.26
4542	AG	S	31	YY	SU	T1	K2	R4	190923	30124	...	0.2725	0.2718	0.26
4543	AG	S	31	YY	SU	T1	K2	R4	190923	30124	...	0.2230	0.2238	0.22

We checked that our classes are balanced, which in this example they are not.

```python
df.groupby(['Cat1'])['Cat2'].count().plot.bar()
```

```
<Axes: xlabel='Cat1'>
```



After we divide our data set into wavenumbers (X) and labels(y), we can use the `imblearn` package which allows to resample the class to match the underespresent class.

```
from imblearn.under_sampling import RandomUnderSampler

rus = RandomUnderSampler(random_state=0)

X_resampled, y_resampled = rus.fit_resample(X, y)
print(sorted(Counter(y_resampled).items()))
```

```
[('AC', 1724), ('AG', 1724)]
```

Also, we need to encode our labels (transform species names into 1 or 0) using `LabelBinarizer`.

```
# Change the labels into 0 or 1
lb = LabelBinarizer()
y_binary = lb.fit_transform(y_resampled)
```

We split our data set into train and test sets. We are going to use our test set for the final evaluation of our ANN model. It is good practice to scale the data, for this we are using standard scaling.

```
# Split into train and test
X_train, X_test, y_train, y_test = train_test_split(X_resampled, y_binary, stratify=y_bina

# Scaling train and test
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

❗ Important

When applying scaling, be sure you fit_transform the train test and only transform the test set. Moreover, scale the data after split it to avoid data leakege.

We further split the training set into training and validation.

```
# Split it further train set into train and validation
X_train_2, X_val, y_train_2, y_val = train_test_split(X_train, y_train, stratify=y_train,
```

One of the things that you can do with keras is the use of callbacks. A callback is an object that can perform actions at various stages of training (e.g. at the start or end of an epoch, before or after a single batch, etc).

You can use callbacks to: Periodically save your model to disk Do early stopping, Get a view on internal states and statistics of a model during training, etc. Here, we are using it to avoid too much print messages during training.

```
# functions to eliminate part of the messages when training ANN
def get_callbacks(name):
  return [
    tfdocs.modeling.EpochDots()
  ]
```

4.1.2 Building the model

We create our the model using the Sequential API as follows:

```
# define the keras model
input_shape = [1796,]
model = keras.Sequential()
model.add(keras.layers.Input(shape=input_shape))
model.add(keras.layers.Dense(500, activation='relu'))
model.add(keras.layers.Dense(1, activation='sigmoid'))
```

Let's go line by line: 1. The first line define the input of the model, which matches how many features (wavenumbers) we have in our data set. 2. `Keras.sequential` will create an object called 'model' in which we can add layers. 3. We add our first layer, the input layer. It has the shape of our previously define input 4. A first hidden layer, with 500 neurons and `'relu'` activation 5. the final output layer with 1 neuron (becuase we are tackling a binary classification) with a `sigmoid` activation

4.1.3 Compiling the model

Once the model is created, we compile it. Here, we defined the loss function (binary crossentropy), the optimizer (adam) and the metric (accuracy) to assess how well our model classifiy our data set.

```
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
```

we can check the model using summary

```
model.summary()
```

```
Model: "sequential"


 Layer (type)                    Output Shape                  Param #

 dense (Dense)                   (None, 500)                   898,500

 dense_1 (Dense)                 (None, 1)                         501


Total params: 899,001 (3.43 MB)
```

```
Trainable params: 899,001 (3.43 MB)

Non-trainable params: 0 (0.00 B)
```

In the following table, you can see what parameters you can start on when building your models depeding on the classification problem.

Table 4.2: Typical classification MLP architecture

Hyperparameter	Regression	Binary classification	Multilabel classification	Multiclass classification
# input neurons	One per input feature	One per input feature	One per input feature	One per input feature
# hidden layers	1 - 5	1 - 5	1 - 5	1 - 5
# neurons per hidden layer	10 - 100	10 - 100	10 - 100	10 - 100
# output neurons	1 per prediction dimension	1	1 per label	1 per class
Hidden activation	reLU	reLU	reLU	reLU
Output activation	None	Logistic	Logistic	Softmax
Loss function	MSE	Cross entropy	Cross entropy	Cross entropy

4.1.4 Training the model

We train the model with `fit()` function. Here, you specify you x and y, number of epochs, batch size and validation data. Also, you can add callbacks

```
history = model.fit(x=X_train_2, y=y_train_2, epochs=500, batch_size=250, validation_data=
```

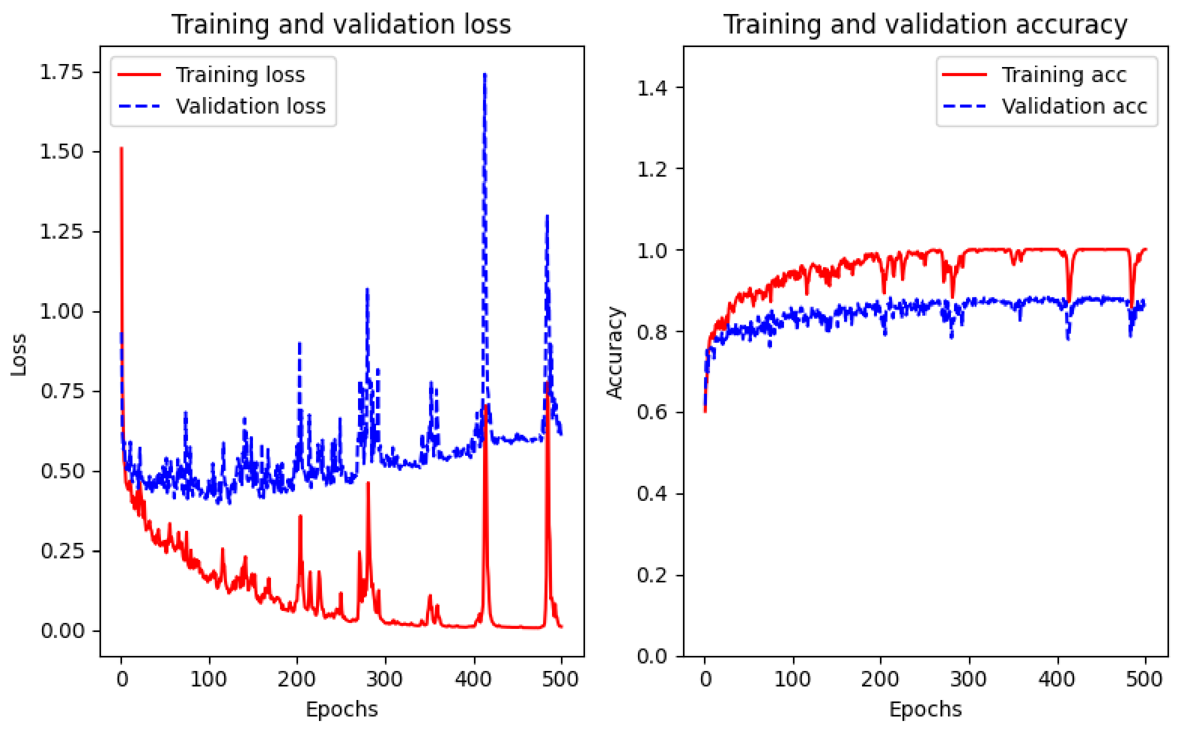
```
Epoch: 0, accuracy:0.6011,  loss:1.5072,  val_accuracy:0.6159,  val_loss:0.9330,
..............................................................................................
Epoch: 100, accuracy:0.9402,  loss:0.1599,  val_accuracy:0.8424,  val_loss:0.4303,
..............................................................................................
Epoch: 200, accuracy:0.9465,  loss:0.1408,  val_accuracy:0.8351,  val_loss:0.5025,
..............................................................................................
Epoch: 300, accuracy:0.9986,  loss:0.0207,  val_accuracy:0.8696,  val_loss:0.5299,
..............................................................................................
```

```
Epoch: 400, accuracy:0.9995,  loss:0.0109,  val_accuracy:0.8750,  val_loss:0.5526,
..............................................................................................
```

We saved all the information of accuracy, loss when using the training set and validation set for each epoch in the `history` variable. You can access the information by using the `history` attribute. A good way to see if your model is overfitting or underfitting is by plotting the values of training/validation loss and accuracy.

```
# Check training and validation curves
fig, (ax, ax2) = plt.subplots(1, 2, figsize=(8,5), tight_layout=True)
loss = history.history['loss']
val_loss = history.history['val_loss']
epochs = range(1, len(loss) + 1)
ax.plot(epochs, loss, 'r-', label='Training loss')
ax.plot(epochs, val_loss, 'b--', label='Validation loss')
ax.set_title('Training and validation loss')
ax.set_xlabel('Epochs')
ax.set_ylabel('Loss')

acc = history.history['accuracy']
val_acc = history.history['val_accuracy']
ax2.plot(epochs, acc, 'r-', label='Training acc')
ax2.plot(epochs, val_acc, 'b--', label='Validation acc')
ax2.set_title('Training and validation accuracy')
ax2.set_xlabel('Epochs')
ax2.set_ylabel('Accuracy')

ax.legend()
ax2.legend()
ax2.set_ylim(0,1.5)
```

```
(0.0, 1.5)
```



Tip

Start we few epochs to see how your computer handles the training. Currently, all these examples are tested on a Macbook pro M1 max pro with 32 GB of ram.

We are not going to discuss overfitting or undefitting in this section yet.

4.1.5 Evaluate with unseen data

We got a pretty decent accuracy with a simple MPL model. Now, we evaluate on the test set

```
# model evaluation
model.evaluate(X_test, y_test)
```

```
22/22                0s 1ms/step - accuracy: 0.8840 - loss: 0.4466
```

```
[0.4332384169101715, 0.8927536010742188]
```

We got an accuracy of 0.89 which is pretty good.

As always, it is good to have confusion matrices to have a better understanding of the model perfomance. Therefore, we make predictions using X_test.

```
# Make predictions
y_proba = model.predict(X_test)
```

```
22/22            0s 2ms/step
```

We round this predictions since they are probailities

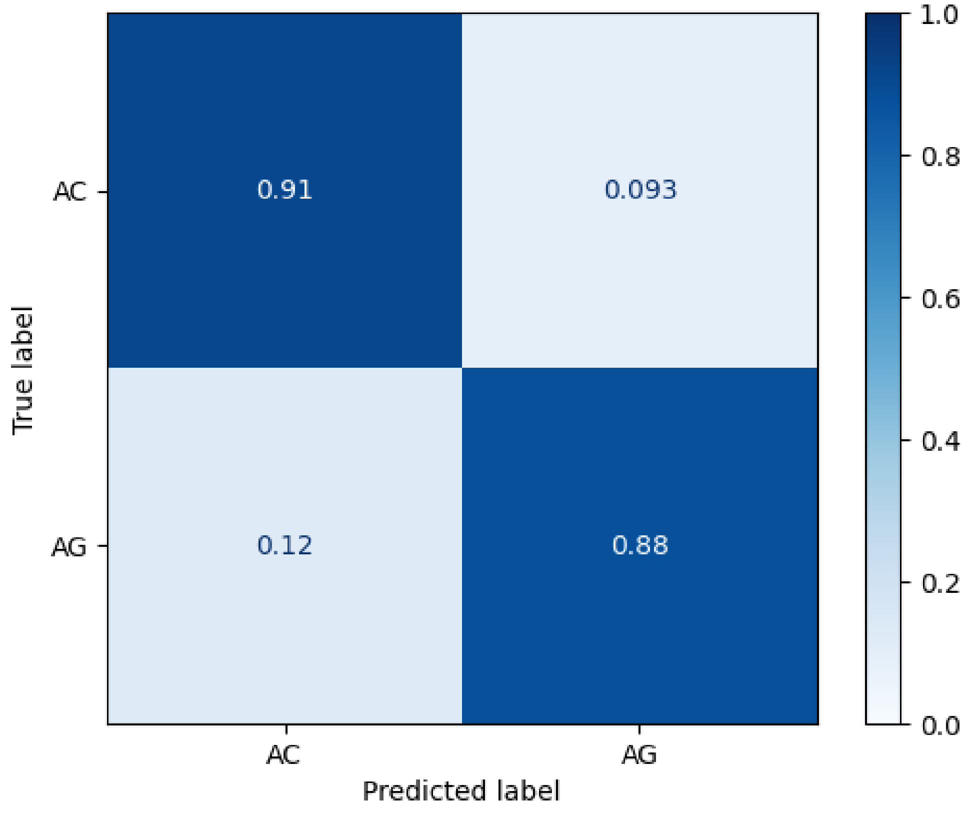
```
# Predictions will be a probability in the range between 0 and 1. So, we round them to hav

y_predicted = y_proba.round(0).astype(int)
y_predicted[0:5]
```

```
array([[1],
       [1],
       [1],
       [1],
       [0]])
```

and then we compute and plot the final confusion matrix

```
# Confusion matrix

from sklearn.metrics import ConfusionMatrixDisplay

ConfusionMatrixDisplay.from_predictions(y_pred=lb.inverse_transform(y_predicted), y_true=l
```

```
<sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at 0x157215fa0>
```



Both classes have quite high accuracy. So we are happy.

4.1.6 Save your model

Training ANN can be quite long and computational expensive. Therefore, it is good practice to save it.

```
# save your model
model.save("./models/my_first_keras_model.keras")
```

You can load your trained model again if you want to continue the project.

```
# load your model
model_new = keras.models.load_model("./models/my_first_keras_model.keras")
```

4.2 A basic artificial neural network for multiclass age classification

The main workflow is exactly the same as for a binary classification. We only need to do some preprocessing to the age classes so they can be read and used bt the neural network.

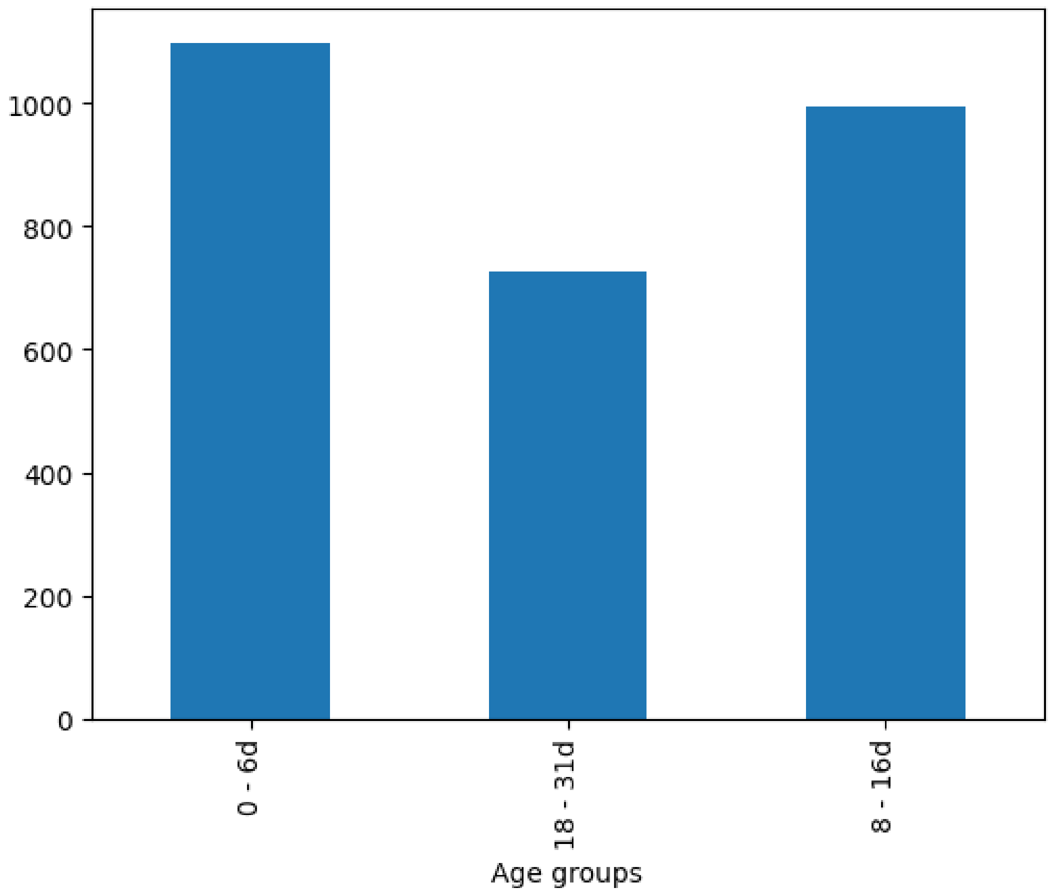
For this example, we subset the original data set using only An. gambiae data. We converted our ages that were numerical into three different categories.

```
df_gambiae['Age groups'].unique()
```

```
array(['0 - 6d', '8 - 16d', '18 - 31d'], dtype=object)
```

The new age grups are not as imbalanced so we will pass. Otherwise, you need to balance the groups

```
<Axes: xlabel='Age groups'>
```



4.2.1 Encoding labels

This is where things get a little more complicated. Keras models do not support labels as it is (you cannot pass the y as categories). Therefore, to make it work we need to encode our labels in a specific way.

First, encode our target labels with values between 0 and n_classes-1. For this, we use `LabelEncoder`

```
# Encode the labels from 0 to n-1 classes
encoder = LabelEncoder()
encoded_y = encoder.fit_transform(y)
encoder.classes_
```

```
array(['0 - 6d', '18 - 31d', '8 - 16d'], dtype=object)
```

Each age group is an integer, `'0 - 6d'` is 0, `'18 - 31d'` is 1 and `'8 - 16d'` is 2.

We need to transform the encoded labels into a binary matrix. This is called OneHotEncoding

```
# transform the encoded labels into a binary class matrix
yhot = utils.to_categorical(encoded_y)
print(yhot)
print(yhot.shape)
```

```
[[1. 0. 0.]
 [1. 0. 0.]
 [1. 0. 0.]
 ...
 [0. 1. 0.]
 [0. 1. 0.]
 [0. 1. 0.]]
(2820, 3)
```

As you can see, each age group has 3 values. For the first sample, the values `[1. 0. 0.]` means that the sample belongs to the group 0 or 0 - 6d. The last sample `[0. 1. 0.]` belongs to the group 1 or 18 - 31d.

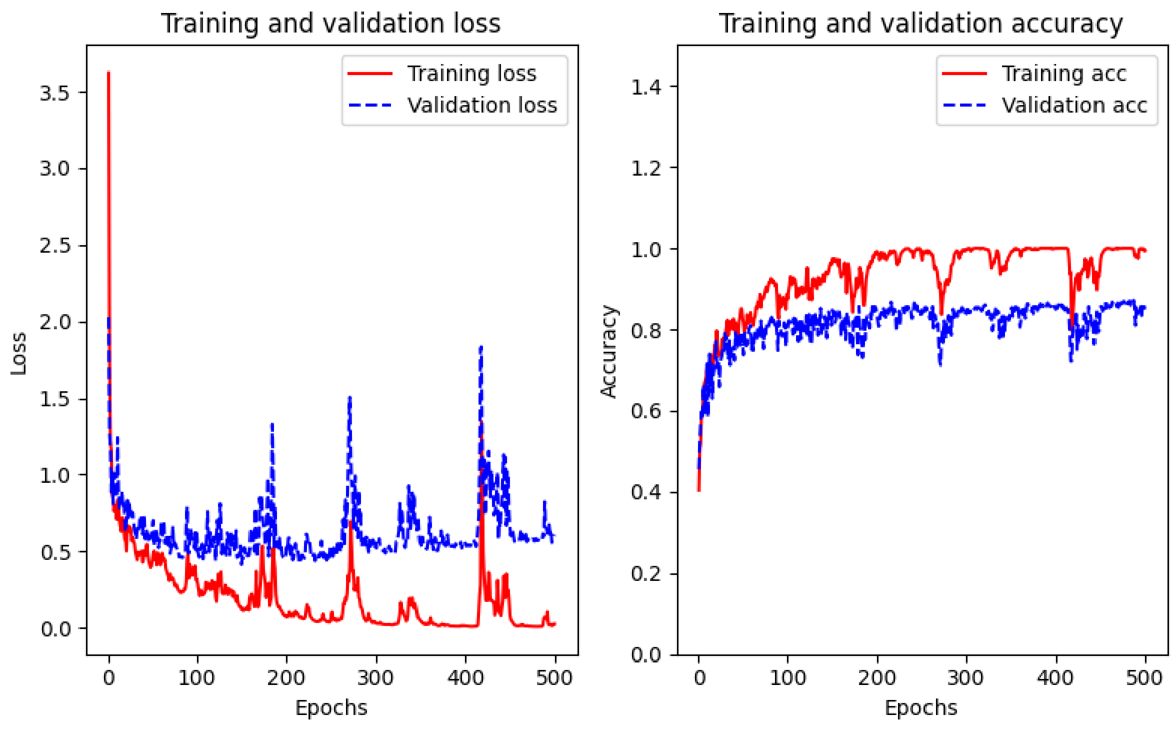
4.2.2 Building the model

For this model we need to change the neurons of the last layer to match the number of labels we have, in this case is 3 and change the activation function to `softmax`. Moreover, we need to use `categorical_crossentropy` as loss function. And that's all

```
input_shape = [1800,]
model = keras.Sequential()
model.add(keras.layers.Input(shape=input_shape))
model.add(keras.layers.Dense(500, activation='relu'))
model.add(keras.layers.Dense(3, activation='softmax'))

# compile the keras model
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
```

After training the model we can see that we got pretty high accuracies.

```
(0.0, 1.5)
```



4.2.3 Evaluate the model with useen data

We evaluate the model using unseen data and we got and accuracy of 82%.

```
18/18              0s 2ms/step - accuracy: 0.8238 - loss: 0.6882
18/18              0s 2ms/step
```

4.2.4 Reverse the encoding

We need to compute the confusion matrix and for that we need to reverse the encoding of our predicted vector and our test vector.

ANN will give us the probability of the sample to belong to each of the 3 categories.

```
# the model gave us the proability of that samples to be 0, 1 or 2 category
print(y_proba[0])
```

```
[1.9061598e-07 6.7195423e-02 9.3280441e-01]
```

Here, we can see that this sample belongs to the category 2 or 8 - 16 d

To change to a vector of 1 dimension with the values of each category before we applied one hot encoding, we use the functon `np.argmax` which will go through each row and give us the index of the highest value.

```
# Select the index with the highest value
y_pred = np.argmax(y_proba, axis=1)
y_pred
```

```
array([2, 2, 1, 2, 0, 0, 0, 2, 0, 2, 2, 0, 2, 1, 1, 0, 0, 1, 1, 2, 2, 0,
       2, 2, 0, 1, 0, 0, 2, 0, 1, 1, 0, 2, 2, 0, 0, 0, 2, 0, 1, 2, 2, 2,
       2, 1, 1, 0, 2, 2, 1, 0, 0, 0, 0, 1, 0, 2, 0, 0, 2, 0, 2, 2, 2, 0,
       2, 2, 0, 0, 2, 0, 2, 0, 0, 2, 2, 1, 0, 2, 0, 2, 2, 2, 2, 1, 0, 2,
       2, 1, 1, 2, 0, 2, 0, 1, 2, 0, 1, 1, 0, 1, 1, 0, 1, 2, 1, 0, 0, 2,
       1, 1, 1, 0, 1, 0, 0, 2, 0, 2, 0, 1, 0, 0, 0, 0, 2, 1, 1, 0, 2, 1,
       0, 1, 2, 1, 2, 0, 0, 0, 2, 2, 1, 0, 1, 2, 2, 0, 2, 2, 0, 1, 1, 1,
       2, 0, 2, 1, 0, 0, 0, 2, 2, 1, 0, 0, 0, 1, 0, 1, 0, 0, 0, 1, 0, 2,
       2, 1, 1, 0, 1, 1, 0, 2, 0, 0, 1, 0, 0, 2, 1, 0, 1, 0, 0, 1, 0, 2,
       2, 0, 0, 0, 2, 2, 1, 1, 0, 0, 1, 1, 2, 0, 1, 2, 0, 0, 2, 2, 2, 0,
       0, 1, 1, 1, 0, 1, 2, 0, 0, 0, 0, 2, 1, 0, 0, 0, 2, 1, 2, 0, 2, 0,
       0, 2, 2, 0, 0, 0, 0, 2, 2, 1, 0, 2, 1, 2, 0, 2, 1, 0, 1, 0, 1, 1,
       2, 2, 0, 0, 0, 2, 2, 0, 2, 0, 1, 0, 0, 1, 0, 0, 1, 2, 1, 1, 1, 0,
       2, 0, 0, 1, 0, 0, 0, 0, 0, 0, 2, 1, 2, 0, 1, 2, 0, 2, 2, 1, 0, 0,
       1, 1, 0, 2, 0, 2, 0, 2, 2, 0, 2, 2, 0, 1, 1, 1, 0, 0, 1, 1, 1, 1,
       1, 2, 1, 2, 1, 2, 0, 0, 2, 2, 2, 2, 0, 2, 1, 1, 1, 1, 2, 2, 0, 0,
       2, 2, 0, 0, 0, 2, 0, 1, 0, 2, 0, 2, 0, 2, 2, 0, 0, 1, 1, 0, 2, 2,
       2, 0, 2, 2, 0, 1, 1, 2, 1, 0, 2, 2, 0, 2, 0, 1, 1, 1, 1, 0, 2, 1,
       0, 0, 1, 2, 0, 2, 0, 1, 2, 0, 0, 0, 2, 0, 2, 0, 2, 2, 0, 0, 2, 0,
       1, 2, 0, 0, 0, 2, 2, 1, 1, 2, 2, 1, 0, 2, 2, 0, 1, 0, 1, 0, 1, 2,
       2, 1, 0, 0, 1, 2, 0, 2, 0, 2, 1, 1, 2, 0, 0, 1, 1, 0, 2, 1, 1, 2,
       1, 0, 1, 1, 0, 1, 2, 0, 0, 0, 1, 1, 0, 0, 0, 2, 0, 0, 2, 1, 1, 2,
       2, 1, 0, 0, 0, 0, 2, 0, 1, 0, 0, 1, 0, 0, 0, 2, 2, 1, 0, 2, 2, 2,
       1, 2, 0, 1, 0, 1, 1, 2, 0, 1, 0, 2, 1, 2, 1, 0, 0, 2, 1, 0, 2, 0,
       1, 0, 0, 0, 2, 2, 1, 1, 2, 2, 1, 0, 2, 0, 0, 2, 0, 1, 2, 2, 2, 2,
       0, 0, 1, 0, 2, 0, 2, 0, 1, 0, 0, 1, 2, 0])
```

You can see that you have a vector with the encoded categories from 0 to 2. We need to pass these indexes to the encoder. There are two ways of doing this, using `encoder.classes_` attribute or the `inverse_transform` function.

```
# reverse the encoding to obtaing the original classes
# one way of doing is using the argument classes from the encoder and pass the indexes of
y_pred_oneform = encoder.classes_[np.argmax(y_proba,axis=1)]


# the second way is use the inverse transform function from the encoder to get the origina
y_pred_secform = encoder.inverse_transform(y_pred)
```

Both approaches give the same results

```
# they are the same
print(y_pred_oneform[0])
print(y_pred_secform[0])
```

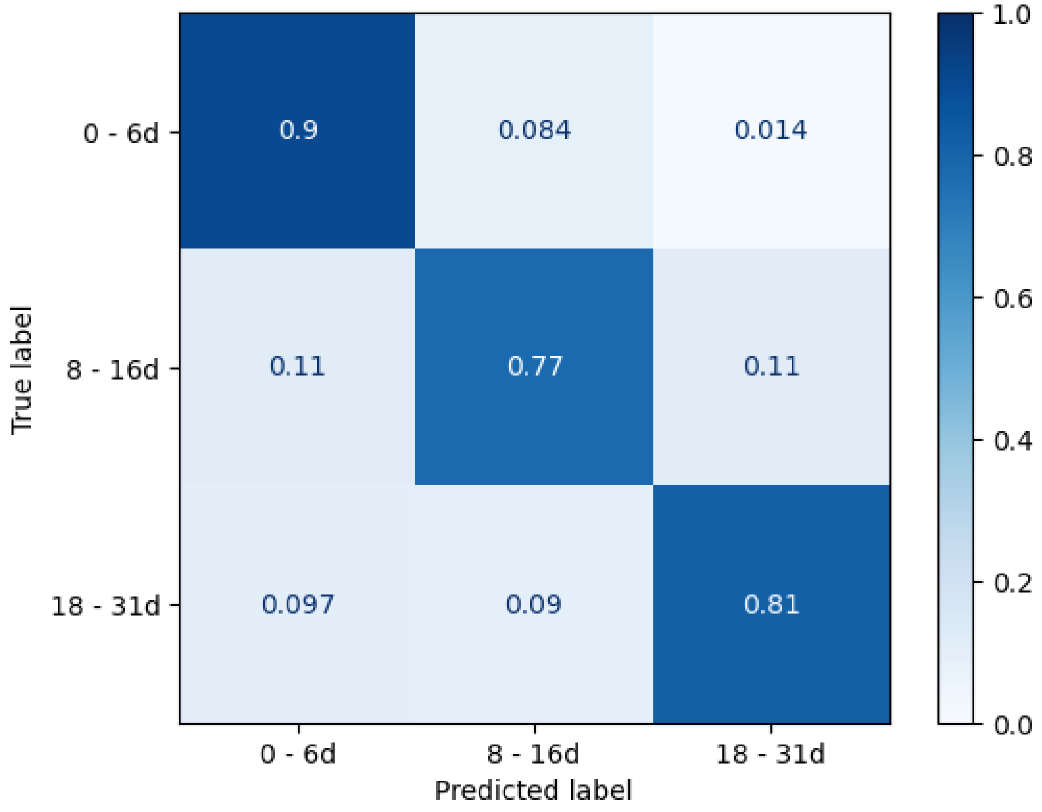
```
8 - 16d
8 - 16d
```

We apply the same process to the y_test and we compute our confusion matrix

```
# Confusion matrix

from sklearn.metrics import ConfusionMatrixDisplay
```

```python
ConfusionMatrixDisplay.from_predictions(
    y_pred=encoder.inverse_transform(y_pred),
    y_true=encoder.inverse_transform(y_new_test),
    normalize='true',
    cmap='Blues',
    im_kw={'vmin':0, 'vmax':1},
    labels=['0 - 6d', '8 - 16d', '18 - 31d'])
```

```
<sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at 0x1047cdfd0>
```



We pass a list of the age groups in the labels parameter to show the correct order of the labels. Otherwise, it the 8 - 16d group will appear last.

if you want to learn more activations check this nice article

The simplest way to prevent overfitting is to start with a small model: A model with a small number of learnable parameters (which is determined by the number of layers and the number of units per layer). In deep learning, the number of learnable parameters in a model is often referred to as the model's "capacity".

Intuitively, a model with more parameters will have more "memorization capacity" and therefore will be able to easily learn a perfect dictionary-like mapping between training samples and their targets, a mapping without any generalization power, but this would be useless when making predictions on previously unseen data.

Always keep this in mind: deep learning models tend to be good at fitting to the training data, but the real challenge is generalization, not fitting.

On the other hand, if the network has limited memorization resources, it will not be able to learn the mapping as easily. To minimize its loss, it will have to learn compressed representations that have more predictive power. At the same time, if you make your model too small, it will have difficulty fitting to the training data. There is a balance between "too much capacity" and "not enough capacity".

Unfortunately, there is no magical formula to determine the right size or architecture of your model (in terms of the number of layers, or the right size for each layer). You will have to experiment using a series of different architectures.

To find an appropriate model size, it's best to start with relatively few layers and parameters, then begin increasing the size of the layers or adding new layers until you see diminishing returns on the validation loss.

Start with a simple model using only densely-connected layers (tf.keras.layers.Dense) as a baseline, then create larger models, and compare them.

4.3 Evaluate your model using crossvalidation.

So far, all the evaluation has been done in a single split of the data sets.Your results will depend on how your data set is split everytime (unless you set a seed). We might want to see how stable is the model with different parts of the data set.

The approach is very similar to ML pipelines section. for this example, we are going to use the whole data set.

4.3.1 Define the cross-validator

First, we need to define our cross-validator. Here, we used `Stratifiedkfold`. The difference between this and Kfold is that Stratifeldkfold maitains the percetage of each class on each fold.

```
sss = StratifiedKFold(n_splits=5,
                      shuffle=True,
                      random_state=123)

# define the callbacks
def get_callbacks(name):
  return [
    tfdocs.modeling.EpochDots()
  ]
```

4.3.2 Cross validation process

Now, the big loop. We create a dictionary called `histories` where we are going to save the results of each fold. We got the first split, and we print the fold number. Clear any model that we have made prviously to save memory (other wise you will get covered of all the models you create on each loop). We build our model, and compile it.

We create a new key in the dictionary `histories` that has the number of the fold and all the results of the fit. This process will happen 5 times. Each time, our model will be trained using for 2000 epochs. This process took 20 minutes in a high end computer so be careful before run the example. It is better if you start with few folds and the epochs beforehand.

```
histories = {}
for i, (train_index, test_index) in enumerate(sss.split(X_resampled, y_binary)):
    print(f"\nFold {i}:")
    keras.backend.clear_session()
    input_shape = [1796,]
    model = keras.Sequential()
    model.add(keras.layers.Input(shape=input_shape))
    model.add(keras.layers.Dense(500, activation='relu'))
    model.add(keras.layers.Dense(1, activation='sigmoid'))

    model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
    histories[f"fold{i}"] = model.fit(x=X_resampled[train_index], y=y_binary[train_index],
```

```
Fold 0:
```

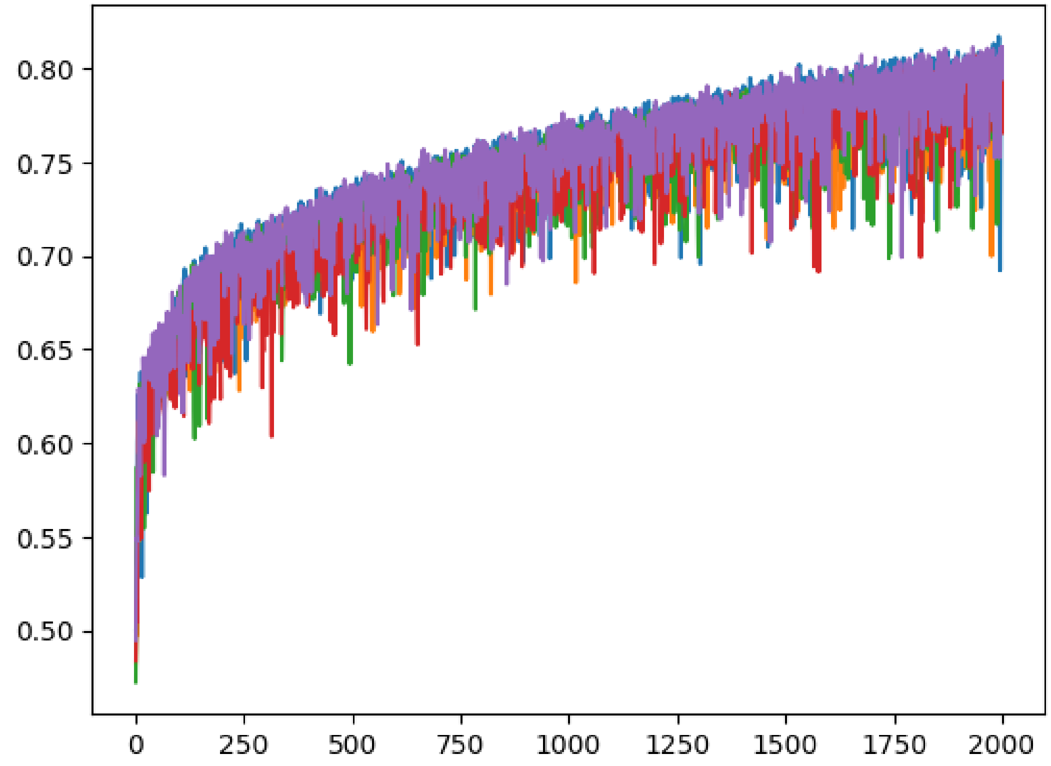
```
Epoch: 0, accuracy:0.5004,  loss:0.7859,  val_accuracy:0.5000,  val_loss:0.7151,
............................................................................................
Epoch: 100, accuracy:0.6302,  loss:0.6369,  val_accuracy:0.6261,  val_loss:0.6385,
............................................................................................
Epoch: 200, accuracy:0.6925,  loss:0.5858,  val_accuracy:0.6826,  val_loss:0.5977,
............................................................................................
Epoch: 300, accuracy:0.6958,  loss:0.5760,  val_accuracy:0.6290,  val_loss:0.6426,
............................................................................................
Epoch: 400, accuracy:0.7288,  loss:0.5394,  val_accuracy:0.7000,  val_loss:0.5660,
............................................................................................
Epoch: 500, accuracy:0.7194,  loss:0.5430,  val_accuracy:0.6870,  val_loss:0.5789,
............................................................................................
Epoch: 600, accuracy:0.7360,  loss:0.5223,  val_accuracy:0.6870,  val_loss:0.5534,
............................................................................................
Epoch: 700, accuracy:0.7509,  loss:0.5021,  val_accuracy:0.7101,  val_loss:0.5307,
............................................................................................
Epoch: 800, accuracy:0.7426,  loss:0.5058,  val_accuracy:0.7333,  val_loss:0.5299,
............................................................................................
Epoch: 900, accuracy:0.7571,  loss:0.4992,  val_accuracy:0.6841,  val_loss:0.5768,
............................................................................................
Epoch: 1000, accuracy:0.7596,  loss:0.4885,  val_accuracy:0.7145,  val_loss:0.5191,
............................................................................................
Epoch: 1100, accuracy:0.7774,  loss:0.4740,  val_accuracy:0.7435,  val_loss:0.5232,
............................................................................................
Epoch: 1200, accuracy:0.7574,  loss:0.4801,  val_accuracy:0.7319,  val_loss:0.5226,
............................................................................................
Epoch: 1300, accuracy:0.7455,  loss:0.5106,  val_accuracy:0.7464,  val_loss:0.4971,
............................................................................................
Epoch: 1400, accuracy:0.7868,  loss:0.4572,  val_accuracy:0.7623,  val_loss:0.4917,
............................................................................................
Epoch: 1500, accuracy:0.7621,  loss:0.4860,  val_accuracy:0.7536,  val_loss:0.4856,
............................................................................................
Epoch: 1600, accuracy:0.7737,  loss:0.4624,  val_accuracy:0.7420,  val_loss:0.4850,
............................................................................................
Epoch: 1700, accuracy:0.7857,  loss:0.4510,  val_accuracy:0.7246,  val_loss:0.5049,
............................................................................................
Epoch: 1800, accuracy:0.7817,  loss:0.4492,  val_accuracy:0.7696,  val_loss:0.4647,
............................................................................................
Epoch: 1900, accuracy:0.7524,  loss:0.5048,  val_accuracy:0.7464,  val_loss:0.4772,
............................................................................................
Fold 1:

Epoch: 0, accuracy:0.4819,  loss:0.7814,  val_accuracy:0.5000,  val_loss:0.7579,
...........................................................................................
Epoch: 100, accuracy:0.6599,  loss:0.6188,  val_accuracy:0.6609,  val_loss:0.6150,
...........................................................................................
Epoch: 200, accuracy:0.6860,  loss:0.5942,  val_accuracy:0.6957,  val_loss:0.5802,
...........................................................................................
Epoch: 300, accuracy:0.6900,  loss:0.5743,  val_accuracy:0.6609,  val_loss:0.6006,
...........................................................................................
Epoch: 400, accuracy:0.7016,  loss:0.5613,  val_accuracy:0.7174,  val_loss:0.5520,
...........................................................................................
Epoch: 500, accuracy:0.7226,  loss:0.5387,  val_accuracy:0.7391,  val_loss:0.5370,
...........................................................................................
Epoch: 600, accuracy:0.7281,  loss:0.5275,  val_accuracy:0.7275,  val_loss:0.5403,
...........................................................................................
Epoch: 700, accuracy:0.7408,  loss:0.5179,  val_accuracy:0.7449,  val_loss:0.5244,
...........................................................................................
Epoch: 800, accuracy:0.7404,  loss:0.5070,  val_accuracy:0.7522,  val_loss:0.5268,
...........................................................................................
Epoch: 900, accuracy:0.7505,  loss:0.5084,  val_accuracy:0.7580,  val_loss:0.5095,
...........................................................................................
Epoch: 1000, accuracy:0.7502,  loss:0.5045,  val_accuracy:0.7087,  val_loss:0.5446,
...........................................................................................
Epoch: 1100, accuracy:0.7165,  loss:0.5448,  val_accuracy:0.7652,  val_loss:0.5109,
...........................................................................................
Epoch: 1200, accuracy:0.7386,  loss:0.5124,  val_accuracy:0.7420,  val_loss:0.5228,
...........................................................................................
Epoch: 1300, accuracy:0.7679,  loss:0.4754,  val_accuracy:0.7696,  val_loss:0.4937,
...........................................................................................
Epoch: 1400, accuracy:0.7451,  loss:0.4951,  val_accuracy:0.7739,  val_loss:0.4920,
...........................................................................................
Epoch: 1500, accuracy:0.7803,  loss:0.4610,  val_accuracy:0.7913,  val_loss:0.4781,
...........................................................................................
Epoch: 1600, accuracy:0.7857,  loss:0.4520,  val_accuracy:0.7899,  val_loss:0.4705,
...........................................................................................
Epoch: 1700, accuracy:0.7806,  loss:0.4511,  val_accuracy:0.7768,  val_loss:0.4811,
...........................................................................................
Epoch: 1800, accuracy:0.7313,  loss:0.5127,  val_accuracy:0.7478,  val_loss:0.5045,
...........................................................................................
Epoch: 1900, accuracy:0.7962,  loss:0.4383,  val_accuracy:0.7913,  val_loss:0.4581,
...........................................................................................
Fold 2:

Epoch: 0, accuracy:0.4724,  loss:0.7904,  val_accuracy:0.5000,  val_loss:0.7427,
...........................................................................................
```

```
Epoch: 100, accuracy:0.6610,  loss:0.6081,  val_accuracy:0.6580,  val_loss:0.6265,
..........................................................................................
Epoch: 200, accuracy:0.7052,  loss:0.5747,  val_accuracy:0.6884,  val_loss:0.6050,
..........................................................................................
Epoch: 300, accuracy:0.7034,  loss:0.5712,  val_accuracy:0.6739,  val_loss:0.6317,
..........................................................................................
Epoch: 400, accuracy:0.7099,  loss:0.5529,  val_accuracy:0.6986,  val_loss:0.5972,
..........................................................................................
Epoch: 500, accuracy:0.7110,  loss:0.5522,  val_accuracy:0.7101,  val_loss:0.5723,
..........................................................................................
Epoch: 600, accuracy:0.7273,  loss:0.5383,  val_accuracy:0.6957,  val_loss:0.5723,
..........................................................................................
Epoch: 700, accuracy:0.7368,  loss:0.5178,  val_accuracy:0.7043,  val_loss:0.5773,
..........................................................................................
Epoch: 800, accuracy:0.7466,  loss:0.5089,  val_accuracy:0.7188,  val_loss:0.5500,
..........................................................................................
Epoch: 900, accuracy:0.7589,  loss:0.5018,  val_accuracy:0.7203,  val_loss:0.5399,
..........................................................................................
Epoch: 1000, accuracy:0.7498,  loss:0.5041,  val_accuracy:0.7246,  val_loss:0.5420,
..........................................................................................
Epoch: 1100, accuracy:0.7498,  loss:0.5002,  val_accuracy:0.7246,  val_loss:0.5344,
..........................................................................................
Epoch: 1200, accuracy:0.7607,  loss:0.4857,  val_accuracy:0.7362,  val_loss:0.5189,
..........................................................................................
Epoch: 1300, accuracy:0.7418,  loss:0.5204,  val_accuracy:0.7420,  val_loss:0.5158,
..........................................................................................
Epoch: 1400, accuracy:0.7495,  loss:0.4984,  val_accuracy:0.7348,  val_loss:0.5235,
..........................................................................................
Epoch: 1500, accuracy:0.7632,  loss:0.4771,  val_accuracy:0.7493,  val_loss:0.5087,
..........................................................................................
Epoch: 1600, accuracy:0.7806,  loss:0.4645,  val_accuracy:0.7623,  val_loss:0.4863,
..........................................................................................
Epoch: 1700, accuracy:0.7161,  loss:0.5561,  val_accuracy:0.7000,  val_loss:0.5782,
..........................................................................................
Epoch: 1800, accuracy:0.7879,  loss:0.4566,  val_accuracy:0.7594,  val_loss:0.4888,
..........................................................................................
Epoch: 1900, accuracy:0.7973,  loss:0.4456,  val_accuracy:0.7507,  val_loss:0.5055,
..........................................................................................
Fold 3:

Epoch: 0, accuracy:0.4839,  loss:0.7824,  val_accuracy:0.5007,  val_loss:0.7118,
..........................................................................................
Epoch: 100, accuracy:0.6549,  loss:0.6249,  val_accuracy:0.6720,  val_loss:0.6153,
...............................................................................................
Epoch: 200, accuracy:0.6796,  loss:0.5972,  val_accuracy:0.6836,  val_loss:0.5846,
...............................................................................................
Epoch: 300, accuracy:0.6970,  loss:0.5768,  val_accuracy:0.6923,  val_loss:0.5728,
...............................................................................................
Epoch: 400, accuracy:0.7010,  loss:0.5695,  val_accuracy:0.7417,  val_loss:0.5383,
...............................................................................................
Epoch: 500, accuracy:0.7068,  loss:0.5523,  val_accuracy:0.7475,  val_loss:0.5231,
...............................................................................................
Epoch: 600, accuracy:0.7249,  loss:0.5344,  val_accuracy:0.7533,  val_loss:0.5101,
...............................................................................................
Epoch: 700, accuracy:0.7340,  loss:0.5314,  val_accuracy:0.7388,  val_loss:0.5127,
...............................................................................................
Epoch: 800, accuracy:0.7082,  loss:0.5589,  val_accuracy:0.7402,  val_loss:0.5101,
...............................................................................................
Epoch: 900, accuracy:0.7470,  loss:0.5109,  val_accuracy:0.7547,  val_loss:0.4902,
...............................................................................................
Epoch: 1000, accuracy:0.7361,  loss:0.5150,  val_accuracy:0.7446,  val_loss:0.5003,
...............................................................................................
Epoch: 1100, accuracy:0.7550,  loss:0.4912,  val_accuracy:0.7576,  val_loss:0.4948,
...............................................................................................
Epoch: 1200, accuracy:0.7361,  loss:0.5083,  val_accuracy:0.7649,  val_loss:0.4848,
...............................................................................................
Epoch: 1300, accuracy:0.7488,  loss:0.4950,  val_accuracy:0.7576,  val_loss:0.4877,
...............................................................................................
Epoch: 1400, accuracy:0.7601,  loss:0.4875,  val_accuracy:0.7808,  val_loss:0.4683,
...............................................................................................
Epoch: 1500, accuracy:0.7760,  loss:0.4711,  val_accuracy:0.7489,  val_loss:0.5041,
...............................................................................................
Epoch: 1600, accuracy:0.7807,  loss:0.4601,  val_accuracy:0.7547,  val_loss:0.5029,
...............................................................................................
Epoch: 1700, accuracy:0.7499,  loss:0.5014,  val_accuracy:0.7358,  val_loss:0.5339,
...............................................................................................
Epoch: 1800, accuracy:0.7938,  loss:0.4431,  val_accuracy:0.7808,  val_loss:0.4482,
...............................................................................................
Epoch: 1900, accuracy:0.7901,  loss:0.4451,  val_accuracy:0.7823,  val_loss:0.4483,
...............................................................................................
Fold 4:

Epoch: 0, accuracy:0.4947,  loss:0.7590,  val_accuracy:0.5007,  val_loss:0.7058,
...............................................................................................
Epoch: 100, accuracy:0.6723,  loss:0.6035,  val_accuracy:0.6720,  val_loss:0.6026,
...............................................................................................
```

```
Epoch: 200, accuracy:0.7035,  loss:0.5731,  val_accuracy:0.6996,  val_loss:0.5966,
............................................................................................
Epoch: 300, accuracy:0.7013,  loss:0.5673,  val_accuracy:0.7126,  val_loss:0.5696,
............................................................................................
Epoch: 400, accuracy:0.7064,  loss:0.5470,  val_accuracy:0.7170,  val_loss:0.5441,
............................................................................................
Epoch: 500, accuracy:0.7311,  loss:0.5306,  val_accuracy:0.6996,  val_loss:0.5380,
............................................................................................
Epoch: 600, accuracy:0.7408,  loss:0.5191,  val_accuracy:0.7344,  val_loss:0.5193,
............................................................................................
Epoch: 700, accuracy:0.7379,  loss:0.5206,  val_accuracy:0.7141,  val_loss:0.5490,
............................................................................................
Epoch: 800, accuracy:0.7608,  loss:0.4969,  val_accuracy:0.7518,  val_loss:0.5097,
............................................................................................
Epoch: 900, accuracy:0.7611,  loss:0.4875,  val_accuracy:0.7736,  val_loss:0.4963,
............................................................................................
Epoch: 1000, accuracy:0.7604,  loss:0.4895,  val_accuracy:0.7678,  val_loss:0.4930,
............................................................................................
Epoch: 1100, accuracy:0.7597,  loss:0.4930,  val_accuracy:0.7489,  val_loss:0.5004,
............................................................................................
Epoch: 1200, accuracy:0.7825,  loss:0.4678,  val_accuracy:0.7736,  val_loss:0.4779,
............................................................................................
Epoch: 1300, accuracy:0.7224,  loss:0.5423,  val_accuracy:0.7736,  val_loss:0.4750,
............................................................................................
Epoch: 1400, accuracy:0.7829,  loss:0.4602,  val_accuracy:0.7025,  val_loss:0.5634,
............................................................................................
Epoch: 1500, accuracy:0.7789,  loss:0.4579,  val_accuracy:0.7881,  val_loss:0.4587,
............................................................................................
Epoch: 1600, accuracy:0.7742,  loss:0.4548,  val_accuracy:0.7170,  val_loss:0.5777,
............................................................................................
Epoch: 1700, accuracy:0.7789,  loss:0.4552,  val_accuracy:0.7837,  val_loss:0.4566,
............................................................................................
Epoch: 1800, accuracy:0.8039,  loss:0.4251,  val_accuracy:0.7358,  val_loss:0.5266,
............................................................................................
Epoch: 1900, accuracy:0.7829,  loss:0.4505,  val_accuracy:0.7983,  val_loss:0.4351,
............................................................................................
```

You can plot the accuracy vs epochs of each fold and see how they are different for each other

```
plt.plot(range(1, len(histories['fold0'].history['accuracy']) + 1), histories['fold0'].his
plt.plot(range(1, len(histories['fold1'].history['accuracy']) + 1), histories['fold1'].his
plt.plot(range(1, len(histories['fold1'].history['accuracy']) + 1), histories['fold2'].his
plt.plot(range(1, len(histories['fold1'].history['accuracy']) + 1), histories['fold3'].his
plt.plot(range(1, len(histories['fold1'].history['accuracy']) + 1), histories['fold4'].his
```



4.3.3 Plotting the mean accuracy and standard deviation

Lets put together all the results in a dataframe

```
acc_total = []
for key in histories.keys():
    acc_df = pd.DataFrame()
    acc_df['val_accuracy'] = pd.DataFrame(histories[key].history['val_accuracy'])
    acc_df['accuracy'] = pd.DataFrame(histories[key].history['accuracy'])
    acc_df['type'] = key
    acc_df['epochs'] = range(1, len(histories[key].history['accuracy']) + 1)
    acc_total.append(acc_df)
```

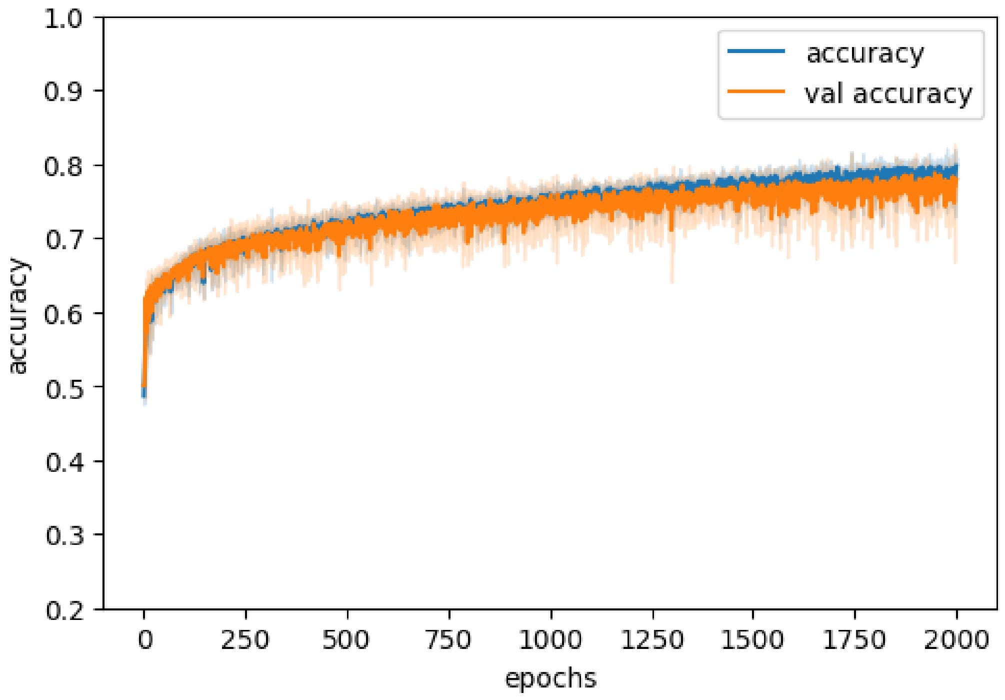
```
df_total = pd.concat(acc_total)
df_total
```

	val_accuracy	accuracy	type	epochs
0	0.500000	0.500363	fold0	1
1	0.498551	0.505801	fold0	2
2	0.518841	0.542422	fold0	3
3	0.543478	0.542785	fold0	4
4	0.620290	0.600435	fold0	5
...	...	...	...	...
1995	0.780842	0.794853	fold4	1996
1996	0.801161	0.802102	fold4	1997
1997	0.770682	0.810801	fold4	1998
1998	0.799710	0.793766	fold4	1999
1999	0.780842	0.811163	fold4	2000

and plot a line with the standard deviation using `seaborn` built-in lineplot function.

```
import seaborn as sns


fig, ax = plt.subplots(figsize=(6,4))
sns.lineplot(data=df_total, x="epochs", y="accuracy", errorbar=('sd', 1), ax=ax, label='ac
sns.lineplot(data=df_total, x="epochs", y="val_accuracy", errorbar=('sd', 1), ax=ax, label
ax.set_ylim(0.2, 1)
```

```
(0.2, 1.0)
```



You can expand this even futher by split again the train set into train and validation set. You can also report the accuracy of your model as the mean plus the standard deviation.

4.4 Tuning a deep learning model

Hyperparameter tuning deep learning models is an art by itself. It will requiere you to run loads of experiements, but also it requieres a deeper understading of your data and model architecture. So, take it easy, it will take time! This example is just a basic pipeline for hyperparameter tuning that once you understand it, you can expand it according to your needs.

We are using keras built in tuner (needs to be installed).

5 XGboost for spectroscopy

5.1 XGboost: A primer

This is a simple pipeline to apply if you want to test XGBoost. The new version of XGboost requieres you to encode your tagets from 0 to n-1 classes.

You need to install the package xgboost using

```
pip install xgboost
```

if you are using conda enviroment use this instead:

```
conda install -c conda-forge py-xgboost
```

Please, check the documentation to learn more.

We follow the similar pipeline of load data and extract our features and targets. For plotting purposes, we extract the name of the columns (wavenumbers) and we save them as integers.

```
df = pd.read_csv("/Users/mauropazmino/Documents/Learning/Deep_learning_tensor/data/MIRS_te

# Extract features and labels
X = np.asarray(df.iloc[:, 15:-1])
y = np.asarray(df.iloc[:,0])

# extract the wavenumbers for plots
wavenumbers = df.iloc[:, 15:-1].columns.astype(int)
```

5.1.1 Encoding our labels

To transform our labels (in this case species) into values of 0 and n-1 classes, we use Label Encoder from sklearn. This process still applies if you have multiple labels.

```
# encoding our labels. XGBoost only accepts values from 0 to n-1 classes

lb = LabelEncoder()
lb.fit(y)
y_encoded = lb.transform(y)
y_encoded
```

```
array([0, 0, 0, ..., 1, 1, 1])
```

5.1.2 Importing the model

We import the model. You will see all the hyperparameters and their default values.

```
from xgboost import XGBClassifier

#define our model
clf = XGBClassifier()
clf
```

```
XGBClassifier(base_score=None, booster=None, callbacks=None,
              colsample_bylevel=None, colsample_bynode=None,
              colsample_bytree=None, device=None, early_stopping_rounds=None,
              enable_categorical=False, eval_metric=None, feature_types=None,
              gamma=None, grow_policy=None, importance_type=None,
              interaction_constraints=None, learning_rate=None, max_bin=None,
              max_cat_threshold=None, max_cat_to_onehot=None,
              max_delta_step=None, max_depth=None, max_leaves=None,
              min_child_weight=None, missing=nan, monotone_constraints=None,
              multi_strategy=None, n_estimators=None, n_jobs=None,
              num_parallel_tree=None, random_state=None, ...)
```

5.1.3 Train and test

We train the model using `fit` function

```
# train
clf.fit(X_train, y_train)
```

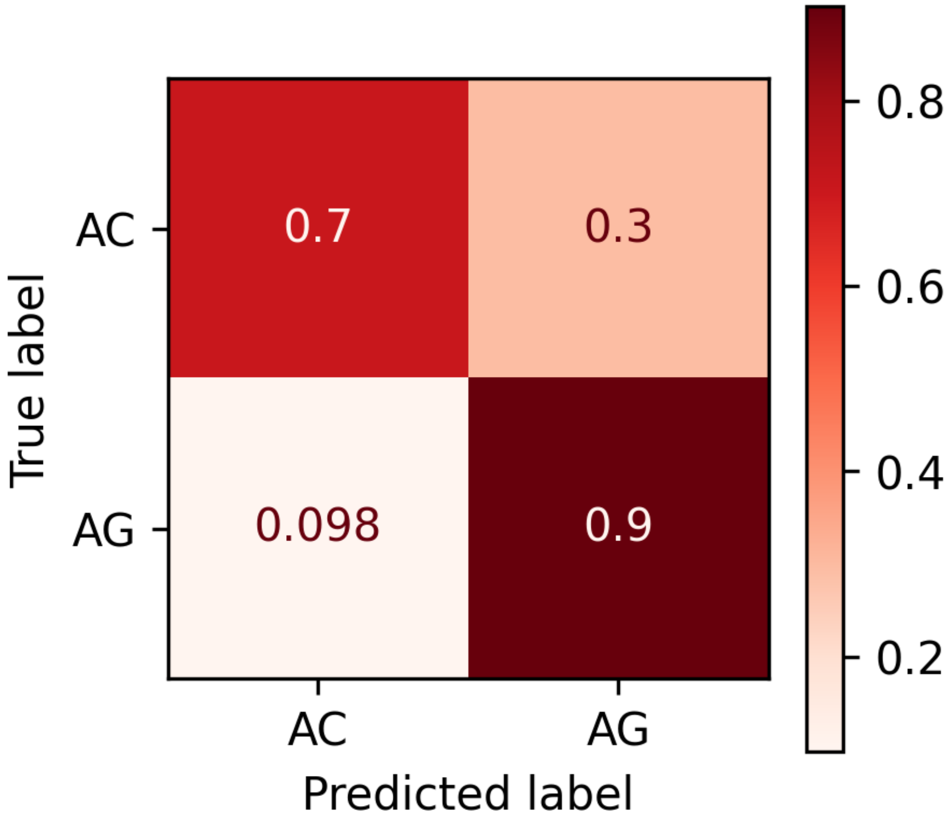
We calculate its accuracy on the test set

```
# predict values
y_pred = clf.predict(X_test)

# accuracy
print(f'Accuracy of XGBoost is: {accuracy_score(y_test, y_pred) * 100:.2f}%')
```

```
Accuracy of XGBoost is: 81.96%
```

and finally a confusion matrix

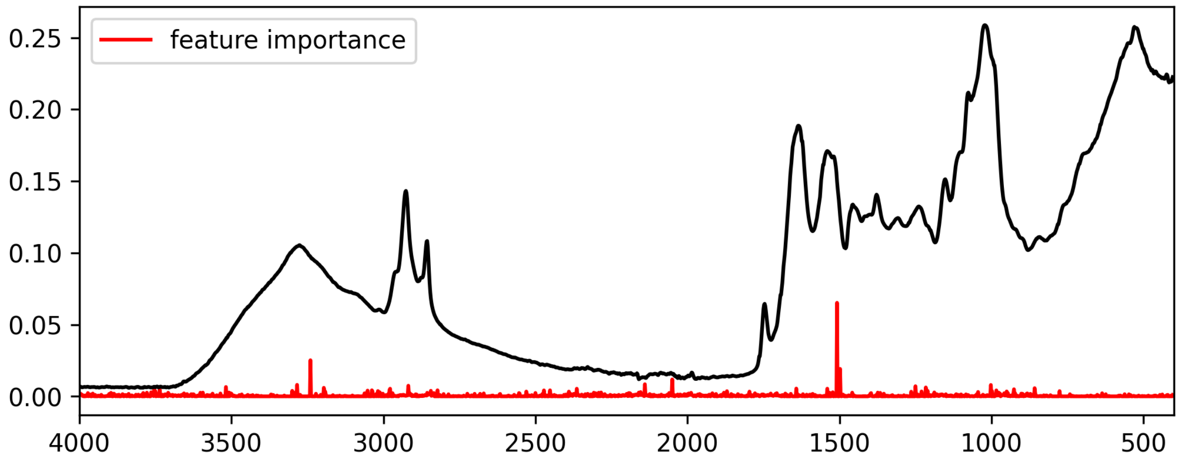
```
fig, ax = plt.subplots(figsize=(3, 3))
ConfusionMatrixDisplay.from_predictions(lb.inverse_transform(y_test), lb.inverse_transform
```



5.1.4 Feature importance

XGBoost have a feature importance attribute that we can use to see which wavenumber has the most influence in the prediction fo species. You can obtain this information with the `feature_importances_` attribute. We plot these values agaist the wavenumbers and superimpose a spectra to have a clear idea of which regions of the spectrum are important.

```
# feature importance
fig, ax = plt.subplots(figsize=(8, 3))
ax.plot(wavenumbers, clf.feature_importances_, color='r', label='feature importance')
ax.plot(wavenumbers, X[1], color='k')
ax.set_xlim(4000, 400)
ax.legend()
```



6 A closer look into spectral quality

The bad blood script created by Dr. Mario Gonzales, assess the spectral quality of a sample based on these three 'metrics'

1. Low intensity
2. Atmospheric interference
3. Distortion by the anvil.

Each one of them can be assessed at different parts of the spectra (Figure 6.1)

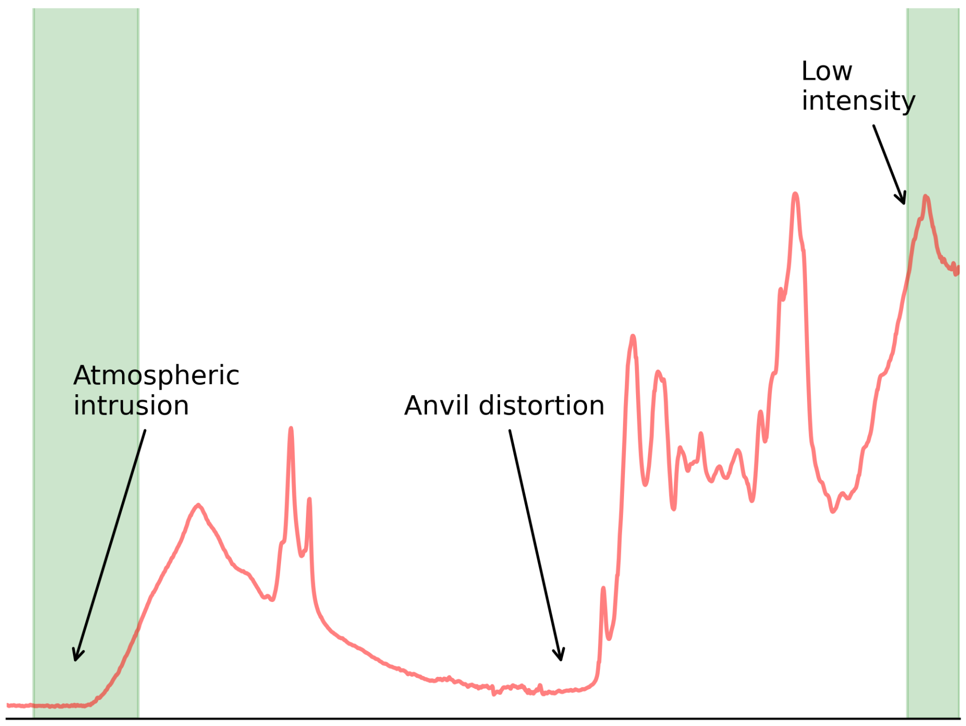


Figure 6.1: The different regions of teh spectra which are used to assess spectral quality on mid infrared

6.1 Low intensity

The mean of the absorbances from 400 to 600 cm^{-1} is used to assses spectral intensity. If the value is less than 0.11, the sample is discarded

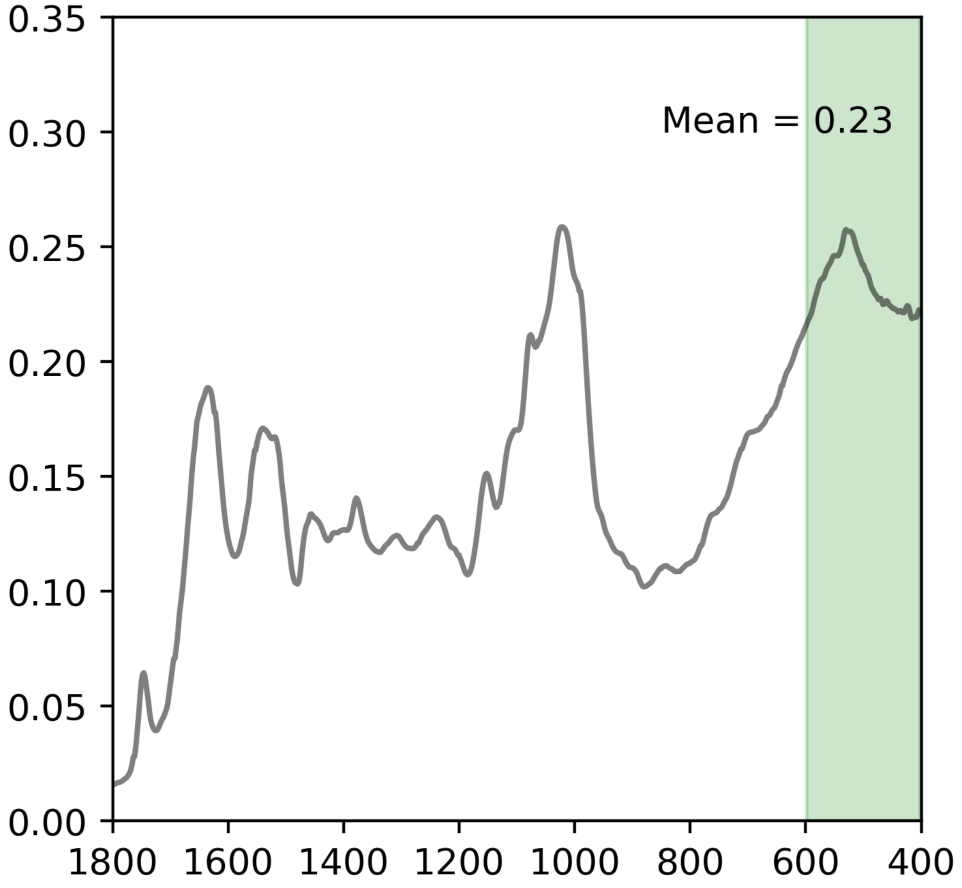


Figure 6.2: The mean of the absorbances from 400 to 600 cm-1 is used to assses spectral intensity. If the value is less than 0.11, the sample is discarded

6.2 Atmospheric intrusion

Water and CO_2 can interfiere with the quality of our measruments. Eventhough, it is crucial to follow thorought protocols to measure samples, error human is always present (samples not well dried specially!). To discard samples that have atmospheric instruion, we assess the region from 3500 to 4000 cm^{-1} due to water can be detected in that region. You can check the regions where water and CO_2 absorb here

First, we fit a polynomial of 5th degree to the region (Figure 6.3). Then, we use R^2 to see how well the polynomial fitted the spectra. If there is no water abosprtion, the line should fit really well, if there is noise due to water, the fit will be bad. We use R^2 as a metric on how well the line fits the spectrum, if R^2 is lower than 0.96, the sample is discarded.

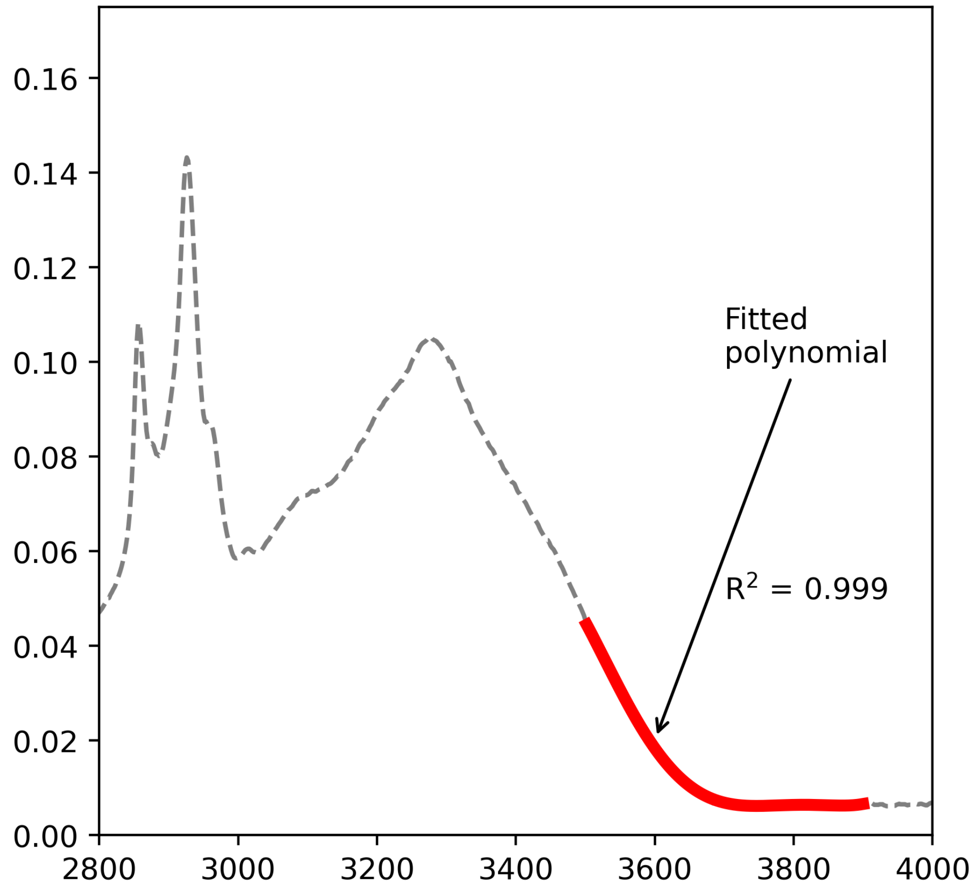


Figure 6.3: 5th degree polynomial fitted to the region 3500 - 3900 cm^{-1}. If R^2 is greater than 0.96, the sample is not discarded

6.3 Anvil distortion

References

González Jiménez, Mario, Simon A. Babayan, Pegah Khazaeli, Margaret Doyle, Finlay Walton, Elliott Reedy, Thomas Glew, et al. 2019. "Prediction of Mosquito Species and Population Age Structure Using Mid-Infrared Spectroscopy and Supervised Machine Learning [Version 3; Peer Review: 2 Approved]." *Wellcome Open Research*. https://doi.org/10.12688/wellcomeopenres.15201.3.

Mwanga, Emmanuel P., Doreen J. Siria, Joshua Mitton, Issa H. Mshani, Mario González-Jiménez, Prashanth Selvaraj, Klaas Wynne, Francesco Baldini, Fredros O. Okumu, and Simon A. Babayan. 2023. "Using Transfer Learning and Dimensionality Reduction Techniques to Improve Generalisability of Machine-Learning Predictions of Mosquito Ages from Mid-Infrared Spectra." *BMC Bioinformatics* 24 (1): 11. https://doi.org/10.1186/s12859-022-05128-5.

Pazmiño-Betancourth, Mauro, Ivan Casas Gómez-Uribarri, Karina Mondragon-Shem, Simon A Babayan, Francesco Baldini, and Lee Rafuse Haines. 2024. "Advancing Age Grading Techniques for Glossina Morsitans Morsitans, Vectors of African Trypanosomiasis, Through Mid-Infrared Spectroscopy and Machine Learning." *Biology Methods and Protocols*, August, bpae058. https://doi.org/10.1093/biomethods/bpae058.

Pazmiño-Betancourth, Mauro, Victor Ochoa-Gutiérrez, Heather M. Ferguson, Mario González-Jiménez, Klaas Wynne, Francesco Baldini, and David Childs. 2023. "Evaluation of Diffuse Reflectance Spectroscopy for Predicting Age, Species, and Cuticular Resistance of Anopheles Gambiae s.l Under Laboratory Conditions." *Scientific Reports* 13 (1): 18499. https://doi.org/10.1038/s41598-023-45696-x.

Siria, Doreen J., Roger Sanou, Joshua Mitton, Emmanuel P. Mwanga, Abdoulaye Niang, Issiaka Sare, Paul C. D. Johnson, et al. 2022. "Rapid Age-Grading and Species Identification of Natural Mosquitoes for Malaria Surveillance." *Nature Communications* 13 (1): 1–9. https://doi.org/10.1038/s41467-022-28980-8.

Vabalas, Andrius, Emma Gowen, Ellen Poliakoff, and Alexander J. Casson. 2019. "Machine Learning Algorithm Validation with a Limited Sample Size." *PLoS ONE* 14 (11): e0224365. https://doi.org/10.1371/journal.pone.0224365.

Varma, Sudhir, and Richard Simon. 2006. "Bias in Error Estimation When Using Cross-Validation for Model Selection." *BMC Bioinformatics* 7 (1). https://doi.org/10.1186/1471-2105-7-91.