**Separating Mouse and Human Antibody Sequences using Protein Encoding and Machine Learning Classifiers**

[ ]:

​

**Questions:**

* How can we train a machine learning classifier to tell apart two groups of protein sequences?
* How can we check for overfitting in a trained machine learning classifier?

**Objectives:**

* To understand how protein sequences can become readable to machine learning predictors.
* To check for overfitted data by testing with a totally naïve dataset.

**Introduction**

If we want to train a classifier that exploits the differences between two groups of proteins, we need to extract numerical information from their constituent amino acid sequences. This is called encoding, and can be done in a number of ways including residue-level encoding of each amino acid with a 1x20 vector, representing the possibility of 20 amino acids at each residue. This is called One Hot Encoding, but often leads to a sparse dataset which is not suitable for machine learning, as each sequence must be spaced out so they are of equal length. Instead, here we use physiochemical properties that may be calculated from the sequence as numeric information (ElAbd *et al.*, 2020).

GB: can use binary encoding with 2\*\*5 or 5 bits just five symbols instead. Why not use both approaches rather than just one? E.g. <https://www.sciencedirect.com/science/article/pii/S2667318522000058>

Antibodies are made up of two heavy and two light chains. However, the functional antigen-binding domains are the Fv, variable fragments, at each fork of the "Y" shape. These Fv fragments are where the variable domains of a heavy chain (VH) and of the light chain (VL) interact. An antibody record is termed ‘paired’ when both the VH and VL sequences of one antibody are known. In the past, this information was rare as it came from studying individual antibodies. However, the advent of B-cell encapsulation and Next Generation Sequencing now allows researchers to sequence a whole repertoire of paired antibodies, and thus this kind of data has become increasingly abundant (Rajan *et al.*, 2018).

In this lesson, we will use a sample of 1000 human and 1000 mouse paired antibodies, obtained from the publicly-available Observed Antibody Space database (Olsen *et al.*, 2022), and train a machine learning classifier to tell them apart. Firstly, we will use Kidera Factors (Nakai *et al.*, 1988) and Propythia (Sequeira *et al.*, 2022) to generate our encodings from an input of FASTA formatted sequences. Secondly, we will split those encodings into training and test datasets for a selection of machine learning classifiers and compare the results for both methods of encoding.

GB: Kidera and Propythia are not explained.

**References**

GB: make links instead of putting links [ElAbd, H., Bromberg, Y., Hoarfrost, A., Lenz, T., Franke, A., & Wendorff, M. (2020). Amino acid encoding for deep learning applications. BMC Bioinformatics, 21(1), 235.](https://doi.org/10.1186/s12859-020-03546-x" \t "_blank)

Nakai, K., Kidera, A., & Kanehisa, M. (1988). Cluster analysis of amino acid indices for prediction of protein structure and function. Protein Engineering, Design and Selection, 2(2), 93-100 <https://doi.org/10.1093/protein/2.2.93>

Olsen, T. H., Boyles, F., & Deane, C. M. (2022). Observed Antibody Space: A diverse database of cleaned, annotated, and translated unpaired and paired antibody sequences. Protein Science, 31(1), 141–146. [https://doi.org/https://doi.org/10.1002/pro.4205](https://doi.org/https:/doi.org/10.1002/pro.4205)

Rajan, S., Kierny, M. R., Mercer, A., Wu, J., Tovchigrechko, A., Wu, H., Dall′Acqua, W. F., Xiao, X., & Chowdhury, P. S. (2018). Recombinant human B cell repertoires enable screening for rare, specific, and natively paired antibodies. Communications Biology, 1(1), 5. <https://doi.org/10.1038/s42003-017-0006-2>

Sequeira, A. M., Lousa, D., & Rocha, M. (2022). ProPythia: A Python package for protein classification based on machine and deep learning. Neurocomputing, 484, 172–182. [https://doi.org/https://doi.org/10.1016/j.neucom.2021.07.102](https://doi.org/https:/doi.org/10.1016/j.neucom.2021.07.102)

## Imports and Requirements

The antibody encoding method we will be using is the Propythia program. Before starting we recommend installing it by copying and pasting the following command in your terminal/command line:

​

GB: it looks like the program below uses Kidera encoding???

​

`pip install propythia`

​

GB: You will also need bio:

​

`pip install bio`

​

The machine learning models that we will use are imported from the [Scikit-Learn](https://scikit-learn.org/stable/) package. We suggest installing this via the following command:

​

`pip install sklearn`

​

As this is a large package, rather than importing the whole package, it is better practise to just import the required functions. Since we need numerous import statements, we will structure these in line with the chronology of the code that we will be using, further into this lesson.

​

[38]:

*##Get Encodings###*

**import** sys

sys.path.append('../src/')

sys.path.append('')

**import** pandas **as** pd

**from** pandas **import** read\_csv

​

**from** propythia.protein.sequence **import** ReadSequence

​

sequence**=**ReadSequence()

​

**from** propythia.protein **import** descriptors

​

*##Data preparation###*

**from** sklearn.utils **import** check\_random\_state, shuffle

**from** sklearn.model\_selection **import** train\_test\_split

**from** sklearn.model\_selection **import** StratifiedShuffleSplit

**from** sklearn.preprocessing **import** StandardScaler

**import** numpy **as** np

**from** numpy **import** pi, linspace, cos, sin, append, ones, zeros, hstack, vstack, intp

**from** numpy **import** mgrid, linspace, c\_, arange, mean, array

**from** numpy.random **import** uniform, seed

​

​

*##Machine Learning Models###*

**from** sklearn.neural\_network **import** MLPClassifier

**from** sklearn.neighbors **import** KNeighborsClassifier

**from** sklearn.svm **import** SVC

**from** sklearn.gaussian\_process **import** GaussianProcessClassifier

**from** sklearn.gaussian\_process.kernels **import** RBF

**from** sklearn.tree **import** DecisionTreeClassifier

**from** sklearn.ensemble **import** RandomForestClassifier, AdaBoostClassifier

**from** sklearn.naive\_bayes **import** GaussianNB

**from** sklearn.discriminant\_analysis **import** QuadraticDiscriminantAnalysis

**from** sklearn.mixture **import** GaussianMixture

**from** sklearn.cluster **import** KMeans

**from** sklearn.cluster **import** DBSCAN

**from** sklearn.cluster **import** SpectralClustering

**from** sklearn.cluster **import** AffinityPropagation

​

​

*##Plotting Results###*

**import** matplotlib.pyplot **as** plt

**from** matplotlib.ticker **import** LinearLocator, FormatStrFormatter

**from** mpl\_toolkits **import** mplot3d

**from** matplotlib.pyplot **import** subplots, axes, scatter, xticks

**from** matplotlib.colors **import** ListedColormap

**from** seaborn **import** heatmap

**from** sklearn.metrics **import** confusion\_matrix

**from** sklearn.metrics **import** ConfusionMatrixDisplay

**from** sklearn.metrics.cluster **import** adjusted\_rand\_score

**from** sklearn **import** metrics

**from** sklearn.metrics **import** matthews\_corrcoef

​

​

*##Model Optimisation###*

**from** sklearn.decomposition **import** PCA

**from** sklearn.model\_selection **import** GridSearchCV

**from** sklearn.metrics **import** make\_scorer

​

GB: there are things that are never used: mplot3d, GridSearchCV

why are they imported?

**Generating the Encoded Dataset**

Here we input our FASTA sequence file and split the entries into VH and VL sequences. We will then run each set of sequences into the Propythia encoder: a dataframe of numerical information for both VH and VL sequences. There are 4000 records in this FASTA file comprising 2000 paired antibodies: 1000 of these are human, and 1000 are mouse.

[25]:

**def** Get\_Kidera\_Encoded\_Sequences(fasta):

VH\_sequences **=** []

VL\_sequences **=** []

Kidera\_Factors **=** {}

**with** open(fasta, "r") **as** f:

**for** line **in** f:

**if** line[0] **==** ">":

**if** "\_VH" **in** line:

sequence\_to\_add **=** f.readline().strip()

VH\_sequences.append(sequence\_to\_add)

**elif** "\_VL" **in** line:

sequence\_to\_add **=** f.readline().strip()

VL\_sequences.append(sequence\_to\_add)

​

**if** len(VH\_sequences) **==** len(VL\_sequences):

VH\_encodings **=** []

VL\_encodings **=** []

**for** i **in** range(len(VH\_sequences)):

VH\_kidera **=** []

**for** res **in** VH\_sequences[i]:

VH\_kidera **=** VH\_kidera **+** Kidera\_Factors[res]

VH\_encodings.append(VH\_kidera)

**for** i **in** range(len(VL\_sequences)):

VL\_kidera **=** []

**for** res **in** VL\_sequences[i]:

VL\_kidera **=** VL\_kidera **+** Kidera\_Factors[res]

VL\_encodings.append(VL\_kidera)

VH\_dataframe **=** pd.DataFrame(VH\_encodings)

VL\_dataframe **=** pd.DataFrame(VL\_encodings)

*# Now we join these two dataframes together so that each sample now has information about its VH and VL sequence.*

VH\_dataframe\_suffix **=** VH\_dataframe.add\_suffix('\_VH')

VL\_dataframe\_suffix **=** VL\_dataframe.add\_suffix('\_VL')

joined\_dataframe\_VH\_VL **=** VH\_dataframe\_suffix.join(VL\_dataframe\_suffix)

joined\_dataframe\_VH\_VL\_cleaned **=** joined\_dataframe\_VH\_VL.dropna(axis**=**1)

**return**(joined\_dataframe\_VH\_VL\_cleaned)

​

[26]:

input\_fasta **=** './HumanMouseOAS\_VH\_VL\_paired\_data.faa'

​

joined\_dataframe\_VH\_VL **=** Get\_Kidera\_Encoded\_Sequences(input\_fasta)

print(joined\_dataframe\_VH\_VL)

0\_VH 1\_VH 2\_VH 3\_VH 4\_VH 5\_VH 6\_VH 7\_VH 8\_VH 9\_VH ... 720\_VL \

0 -0.47 0.24 0.07 1.10 1.10 0.59 0.84 -0.71 -0.03 -2.33 ... -1.56

1 -0.47 0.24 0.07 1.10 1.10 0.59 0.84 -0.71 -0.03 -2.33 ... -1.04

2 -1.45 0.19 -1.61 1.17 -1.31 0.40 0.04 0.38 -0.35 -0.12 ... 0.26

3 -1.45 0.19 -1.61 1.17 -1.31 0.40 0.04 0.38 -0.35 -0.12 ... -1.56

4 -0.47 0.24 0.07 1.10 1.10 0.59 0.84 -0.71 -0.03 -2.33 ... 1.46

... ... ... ... ... ... ... ... ... ... ... ... ...

1995 -0.47 0.24 0.07 1.10 1.10 0.59 0.84 -0.71 -0.03 -2.33 ... 1.46

1996 -1.45 0.19 -1.61 1.17 -1.31 0.40 0.04 0.38 -0.35 -0.12 ... 0.81

1997 -0.47 0.24 0.07 1.10 1.10 0.59 0.84 -0.71 -0.03 -2.33 ... -1.04

1998 -0.47 0.24 0.07 1.10 1.10 0.59 0.84 -0.71 -0.03 -2.33 ... -1.56

1999 -1.45 0.19 -1.61 1.17 -1.31 0.40 0.04 0.38 -0.35 -0.12 ... 0.81

721\_VL 722\_VL 723\_VL 724\_VL 725\_VL 726\_VL 727\_VL 728\_VL 729\_VL

0 -1.67 -0.97 -0.27 -0.93 -0.78 -0.20 -0.08 0.21 -0.48

1 0.00 -0.24 -1.10 -0.55 -2.05 0.96 -0.76 0.45 0.93

2 -0.70 1.21 0.63 -0.10 0.21 0.24 -1.15 -0.56 0.19

3 -1.67 -0.97 -0.27 -0.93 -0.78 -0.20 -0.08 0.21 -0.48

4 -1.96 -0.23 -0.16 0.10 -0.11 1.32 2.36 -1.66 0.46

... ... ... ... ... ... ... ... ... ...

1995 -1.96 -0.23 -0.16 0.10 -0.11 1.32 2.36 -1.66 0.46

1996 -1.08 0.16 0.42 -0.21 -0.43 -1.89 -1.15 -0.97 -0.23

1997 0.00 -0.24 -1.10 -0.55 -2.05 0.96 -0.76 0.45 0.93

1998 -1.67 -0.97 -0.27 -0.93 -0.78 -0.20 -0.08 0.21 -0.48

1999 -1.08 0.16 0.42 -0.21 -0.43 -1.89 -1.15 -0.97 -0.23

[2000 rows x 1780 columns]

[ ]:

GB: what do the numbers mean**?** This needs some explanation

Now we have our encodings and we need to prepare our labels. As our input was in the order: 1000 Human antibodies and 1000 Mouse antibodies, we can simply make a list showing only these:

[30]:

*#Prepare training data and labels*

labels1 **=** 1000**\***[1] *##Human antibodies will be class 1*

labels2 **=** 1000**\***[0] *## Mouse antibodies will be class 0*

labels **=** labels1**+**labels2

​

y**=**labels

*##Mouse ==1, Human == 0*

​

dataset **=** joined\_dataframe\_VH\_VL

dataset**=**dataset.loc[:, dataset.columns **!=** 'Unnamed: 0']

print(len(y) **==** dataset.shape[0]) *## Check whether number of labels and number of samples are the same*

​

True

Now that we have our datasets, we may split them into training datasets for fitting and testing datasets to verify their effectiveness as predictors. A 70/30 or 80/20 split can be used.

Define RANDOM\_SEED here (in the code – it gives an error).

[34]:

X\_train, X\_test, y\_train, y\_test **=** train\_test\_split(dataset, y, test\_size**=**.3, random\_state**=**RANDOM\_SEED, shuffle**=True**)

​

num\_rows, num\_cols **=** dataset.shape

​

print("Training set: ", X\_train.shape, " Test set: ", X\_test.shape)

​

Training set: (1400, 1780) Test set: (600, 1780)

**Separating our data with Machine Learning Classifiers**

Next, we can import the classifiers for testing:

[35]:

n**=**2

​

RANDOM\_SEED**=**42

​

classifiers **=** {

'SVC':SVC(kernel**=**"linear", C**=**0.025),

'SVC2': SVC(gamma**=**2, C**=**1),

'DecisionTree': DecisionTreeClassifier(max\_depth**=**5),

'RFC': RandomForestClassifier(max\_depth**=**5, n\_estimators**=**10, max\_features**=**1),

'MLPC': MLPClassifier(alpha**=**1, max\_iter**=**1000, random\_state**=**42),

'ADABoost':AdaBoostClassifier(),

'GaussianNB': GaussianNB(),

'QDA':QuadraticDiscriminantAnalysis(),

}

The next step is to loop over our classifiers and use the test and train datasets to generate a score to validate the classifiers. We have chosen the Matthews Correlation Coefficient (MCC) which is a metric less prone to bias, as it accounts for both false predictions, and true predictions. This metric lies on a score between -1 (inverse prediction) and 1 (perfect prediction), with 0 being coin-toss likelihood (random labelling).

GB: How is the metrix defined? Why not other metrics as well?

We then plot the result as a confusion matrix, demonstrating the predictive power of our classifiers. The confusion matrix shows the raw number of records that have been assigned to each category in a 2x2 matrix and is given as such:

|  | **Predicted Class = 0** | **Predicted Class = 1** |
| --- | --- | --- |
| Actual Class = 0 | True Negative | False Positive |
| Actual Class = 1 | False Negative | True Positive |

Ideally we want the True Negative and True Positive field to be the most popular fields, with none or few records in the False Positive and False Negative fields.

Note that results may vary between each run due to the stochastic nature of the machine learning algorithms.

[68]:

*##Loop through each classifier, fit training data and evaluate model. Plot results as confusion matrix##*

​

no\_classifiers **=** len(classifiers)

​

scores **=** []

​

confusion\_matrices **=** zeros((2, 2, no\_classifiers))

​

**for** index, classifier **in** enumerate(classifiers):

clf **=** classifiers.get(classifier)

clf.fit(X\_train,y\_train)

y\_predict **=** clf.predict(X\_test)

scoring **=** matthews\_corrcoef(y\_test, y\_predict)

scores.append(scoring)

confusion\_matrices[:, :, index] **=** metrics.confusion\_matrix(y\_test, y\_predict)

*# heatmap(confusion\_matrix, annot=True, cmap='summer', ax=ax[index])*

​

​

print('Complete')

/Users/geroldbaier/anaconda3/lib/python3.11/site-packages/sklearn/ensemble/\_weight\_boosting.py:519: FutureWarning: The SAMME.R algorithm (the default) is deprecated and will be removed in 1.6. Use the SAMME algorithm to circumvent this warning.

warnings.warn(

/Users/geroldbaier/anaconda3/lib/python3.11/site-packages/sklearn/discriminant\_analysis.py:935: UserWarning: Variables are collinear

warnings.warn("Variables are collinear")

GB: figures are horrible. 8 separate confusion matrices cannot be viewed at the same time.

[118]:

fig, ax **=** subplots(nrows**=**no\_classifiers, figsize**=**(8, 16))

​

**for** index, classifier **in** enumerate(classifiers):

​

disp **=** ConfusionMatrixDisplay(confusion\_matrix**=**confusion\_matrices[:, :, index], display\_labels**=**clf.classes\_)

disp.plot(ax**=**ax[index], cmap**=**'summer');

ax[index].set\_title(classifier)

​

fig.tight\_layout()

​

plt.show()

A green and yellow squares with black text

Description automatically generated

[1]:

*##Plot Performance of all Models##*

fig, ax **=** plt.subplots(figsize**=**(8, 4))

plt.suptitle('Classifier Performance', fontsize**=**16)

​

bins **=** arange(len(classifiers))

ax.bar(arange(len(scores)), scores)

ax.set\_ylabel('Matthews Correlation Coefficient')

ax.set\_xlabel('Classifiers', fontsize**=**10)

ax.set\_xticks(bins)

ax.set\_xticklabels(classifiers, rotation**=-**40);

​

print(np.around(scores, 2))

As we can see, the majority of these classifiers have successfully learned to classify the two groups of protein structure, with nearly perfect performance in the SVC model, the neural network MLPC and ADABoost. It is evident that there are highly correlated features in the training data which the models have exploited. However, the Kidera factor is not the only method of encoding sequences for machine learning algorithms. In fact, for more complex problems, deeper methods of encoding may be required.

**Using Amino Acid Compositions to Encode Protein Sequences**

Amino acid compositions are a sequence-based statistic as compared to the quantitative physico-chemical properties evaluated in the Kidera factors. The former generates much denser, more complex encodings, which we will compare with the Kidera factor method.

[120]:

*###Propythia Command to get encodings###*

**def** get\_descriptors(protein):

*##This will retrieve a selection of encodings that are not dependent on the sequence length##*

test**=** protein.adaptable([3,4,5,6,7,8,9,10,11,12,13,14,17,18,19,20,21])

**return**(test)

[121]:

**I’d say the code here needs careful commenting to explain each individual subsection, loop, etc.**

**def** Get\_dataset(fasta):

VH\_sequences **=** []

VL\_sequences **=** []

**with** open(fasta, "r") **as** f:

**for** line **in** f:

**if** line[0] **==** ">":

**if** "\_VH" **in** line:

sequence\_to\_add **=** f.readline().strip()

VH\_sequences.append(sequence\_to\_add)

**elif** "\_VL" **in** line:

sequence\_to\_add **=** f.readline().strip()

VL\_sequences.append(sequence\_to\_add)

​

print(len(VH\_sequences),len(VL\_sequences))

**if** len(VH\_sequences) **==** len(VL\_sequences):

VH\_dataframe **=** pd.DataFrame()

VL\_dataframe **=** pd.DataFrame()

**for** i **in** range(len(VH\_sequences)):

ps\_string**=**sequence.read\_protein\_sequence(VH\_sequences[i])

protein **=** Descriptor(ps\_string)

descriptors **=** get\_descriptors(protein)

*#VH\_dataframe.loc[len(VH\_dataframe)] = descriptors*

VH\_dataframe **=** VH\_dataframe.\_append(descriptors, ignore\_index**=True**)

print("VH\_data", VH\_dataframe.shape)

**for** i **in** range(len(VL\_sequences)):

ps\_string**=**sequence.read\_protein\_sequence(VL\_sequences[i])

protein **=** Descriptor(ps\_string)

descriptors **=** get\_descriptors(protein)

*#VL\_dataframe.loc[len(VL\_dataframe)] = descriptors*

VL\_dataframe **=** VL\_dataframe.\_append(descriptors, ignore\_index**=True**)

print("VL\_data", VL\_dataframe.shape)

*# Now we join these two dataframes together so that each sample now has information about its VH and VL sequence.*

VH\_dataframe\_suffix **=** VH\_dataframe.add\_suffix('\_VH')

VL\_dataframe\_suffix **=** VL\_dataframe.add\_suffix('\_VL')

joined\_dataframe\_VH\_VL **=** VH\_dataframe\_suffix.join(VL\_dataframe\_suffix)

**return**(joined\_dataframe\_VH\_VL)

​

[122]:

*### Input Fasta and Run Dataset###*

input\_fasta **=** './HumanMouseOAS\_VH\_VL\_paired\_data.faa'

​

*#joined\_dataframe\_VH\_VL = Get\_dataset(input\_fasta)*

​

*#Optionally save dataframe as a CSV to simply reload it in future*

*#joined\_dataframe\_VH\_VL.to\_csv('./HumanMouseOAS\_VH\_VL\_paired\_data.faa\_Full\_descriptors')#*

joined\_dataframe\_VH\_VL **=** read\_csv('./HumanMouseOAS\_VH\_VL\_paired\_data.faa\_Full\_descriptors.csv', header **=** 0)

**Do it yourself**

* These Propythia encodings were selected to reduce the time taken to run. Retry the encoding step and experiment with the protein.adaptable([3,4,5,6,7,8,9,10,11,12,13,14,17,18,19,20,21]) array.
* Note that Propythia accepts numbers from 0-40, however we avoid 1, 2 and 37 as these produce outputs of differing length.

[123]:

RANDOM\_SEED **=** 0

*#Prepare training data and labels*

labels1 **=** 1000**\***[1] *##Human antibodies will be class 1*

labels2 **=** 1000**\***[0] *## Mouse antibodies will be class 0*

labels **=** labels1**+**labels2

y**=**labels

print(len(y))

*##Mouse ==1, Human == 0*

​

dataset **=** joined\_dataframe\_VH\_VL

dataset**=**dataset.loc[:, dataset.columns **!=** 'Unnamed: 0']

print(dataset.shape) *##Just to check that you have an equal number of labels to the number of samples*

​

2000

(2000, 890)

[124]:

X\_train, X\_test, y\_train, y\_test **=** train\_test\_split(dataset, y, test\_size**=**.3, random\_state**=**RANDOM\_SEED, shuffle**=True**)

num\_rows, num\_cols **=** dataset.shape

print("Training set size: ", X\_train.shape, " Test set size: ", X\_test.shape)

​

Training set size: (1400, 890) Test set size: (600, 890)

**Do it yourself**

* The list of classifiers used above is not exhaustive. These were picked to represent the major classes of models. You will find a much larger list of classifiers supported in scikit-learn, here: <https://scikit-learn.org/stable/supervised_learning.html>.
* Try adding some new classifiers to the classifier dictionary and check how the results differ.

GB: this is pretty much the same thing as above. Is it not enough to name the classifiers and give the scores as an answer?

[134]:

RANDOM\_SEED**=**42

​

classifiers **=** {

'KNeighbours': KNeighborsClassifier(2),

'Gaussian': GaussianMixture(n\_components**=**n),

'KMeans': KMeans(n\_clusters**=**n)

}

​

GB: what about these? These are for clustering

[135]:

*##Loop through each classifier, fit training data and evaluate model. Plot results as confusion matrix##*

​

no\_classifiers **=** len(classifiers)

​

scores **=** []

​

confusion\_matrices **=** zeros((2, 2, no\_classifiers))

​

**for** index, classifier **in** enumerate(classifiers):

clf **=** classifiers.get(classifier)

clf.fit(X\_train,y\_train)

y\_predict **=** clf.predict(X\_test)

scoring **=** matthews\_corrcoef(y\_test, y\_predict)

scores.append(scoring)

confusion\_matrices[:, :, index] **=** metrics.confusion\_matrix(y\_test, y\_predict)

*# heatmap(confusion\_matrix, annot=True, cmap='summer', ax=ax[index])*

​

​

print('Complete')

Complete

[136]:

fig, ax **=** subplots(nrows**=**no\_classifiers, figsize**=**(8, 16))

​

**for** index, classifier **in** enumerate(classifiers):

​

print(classifier)

**if** classifier **==** 'KMeans':

disp **=** ConfusionMatrixDisplay(confusion\_matrix**=**confusion\_matrices[:, :, index], display\_labels**=**clf.labels\_)

**else**:

disp **=** ConfusionMatrixDisplay(confusion\_matrix**=**confusion\_matrices[:, :, index], display\_labels**=**clf.classes\_)

​

​

disp.plot(ax**=**ax[index], cmap**=**'summer');

​

ax[index].set\_title(classifier)

​

​

fig.tight\_layout()

​

plt.show()

KNeighbours

---------------------------------------------------------------------------

AttributeError Traceback (most recent call last)

Cell In[136], line 9

**7** disp = ConfusionMatrixDisplay(confusion\_matrix=confusion\_matrices[:, :, index], display\_labels=clf.labels\_)

**8** **else**:

----> 9 disp = ConfusionMatrixDisplay(confusion\_matrix=confusion\_matrices[:, :, index], display\_labels=clf.classes\_)

**12** disp.plot(ax=ax[index], cmap='summer');

**14** ax[index].set\_title(classifier)

AttributeError: 'KMeans' object has no attribute 'classes\_'

A graph of a graph

Description automatically generated with medium confidence

[125]:

*# ##Loop through each classifier, fit training data and evaluate model. Plot results as confusion matrix##*

*# scores = []*

​

*# for i in classifiers:*

*# clf\_1 = classifiers.get(i)*

*# clf\_1.fit(X\_train,y\_train)*

*# y\_predict1 = clf\_1.predict(X\_test)*

*# scoring = matthews\_corrcoef(y\_test, y\_predict1)*

*# scores.append(scoring)*

*# confusion\_matrix = metrics.confusion\_matrix(y\_test, y\_predict1)*

*# ax1 = sns.heatmap(confusion\_matrix, annot=True, cmap='summer')*

*# title = str(i)*

*# ax1.set\_title(title);*

*# ax1.set\_xlabel('\nPredicted Values')*

*# ax1.set\_ylabel('Actual Values ');*

*# plt.show()*

​

​

A chart with green squares and yellow squares

Description automatically generated

A chart with green squares and yellow squares

Description automatically generated

A green and yellow squares with white text

Description automatically generated

A green and yellow squares with numbers

Description automatically generated

A chart with green and yellow squares

Description automatically generated

/Users/geroldbaier/anaconda3/lib/python3.11/site-packages/sklearn/ensemble/\_weight\_boosting.py:519: FutureWarning: The SAMME.R algorithm (the default) is deprecated and will be removed in 1.6. Use the SAMME algorithm to circumvent this warning.

warnings.warn(

A chart of a green and yellow box

Description automatically generated with medium confidence

A green and yellow squares with black numbers

Description automatically generated

/Users/geroldbaier/anaconda3/lib/python3.11/site-packages/sklearn/discriminant\_analysis.py:935: UserWarning: Variables are collinear

warnings.warn("Variables are collinear")

A green and yellow squares with white text

Description automatically generated

[126]:

*##Plot Performance of all Models##*

fig, ax **=** plt.subplots(figsize**=**(8,6))

plt.suptitle('Performance of Machine Learning Classifiers Against Mouse and Human Antibodies', fontsize**=**20)

​

bins **=** arange(len(classifiers))

ax.bar(arange(len(scores)), scores)

ax.set\_ylabel('Matthews Correlation Coefficient')

ax.set\_xlabel('Classifiers')

ax.set\_xticks(bins)

ax.set\_xticklabels(classifiers, rotation**=-**80);

A graph of blue bars

Description automatically generated

The above chart indicates that the best performing predictors are ADA\_Boost, GaussianNB, DecisionTree and SVC, whereas both Gaussian and KMeans are the worst performing with negative MCC scores.

GB: ADABoost is without underscore, it should be GaussianNB; and naive Bayesian needs explanation. So many typos. Where is Gaussian (GMM?) and KMeans in the graphic???

**Questions**

* Which encoding method produced the best results, overall? Why could this be? *Hint: think of the size of both sets of encodings.*
* What could be the advantages and disadvantages of both sets of encodings?

GB: If people have no background in how the algorithms work, they cannot answer the "why could this be" question.

**End of Chapter Exercise: Testing our Classifiers on a Naïve Dataset**

We have seen that it is possible to separate mouse and human antibody protein sequences through their numerical encodings.

We can also take a totally naïve dataset that the model has not been exposed to. This is a measure we can take to check for overfitting. If we see that there is poor performance on this naïve "held back" dataset, then it could suggest overfitting to the training data. Using 20 human and 20 mouse paired sequences (not previously used to train our models) from the OAS database, it is possible to generate their encodings, and pass them through the optimised model, in order to test it.

The file below has 20 human and 20 mouse sequences, which are held back from our original training data. Using the skills you have learned so far in this notebook, encode these paired sequences and generate a list of labels for these entries. Pass them through the trained classifiers and evaluate their performance.

* Comment on which classifier performs best.
* Think of ways in which the classifiers can be improved. These will be expanded upon in the next notebook.

naive\_fasta = './Naive\_dataset.faa.txt'

**Keypoints**

* Protein sequences must be numerically encoded to be readable by machine learning algorithms.
* It is necessary to test your classifier with multiple models and multiple encoding methods to find which works best.