Machine Learning Project

Stage 2

I. Introduction:

The genome of an organism contains 64 codons, which are triplets of the 4 nucleotides A, T, G, and C. These 64 codons are mapped to roughly 20 amino acids, which are the building blocks of protein. An organism's codon usage therefore carries information about its protein profile, but it goes beyond that – due to the inherent redundancy of genetic code, the same amino acid can be encoded by different (synonymous) codons, and preferences in the usage of synonymous codons may also reveal a lot of interesting information about an organism's taxonomic and phylogenetic features.

This machine learning project aims to utilize codon usage information of nucleic acid molecules to predict the taxonomic domain of the organism to which the nucleic acid belongs. This project works as an additional proof-of-concept for the close connection between an organism's genetic code and its taxonomic classification.

In section II, we will first formalize our machine learning problem and give an overview of the dataset. Section III discusses the methods applied in different stages such as data preprocessing, feature selection, model choice, and model validation. The results obtained are discussed in section IV. In section V, we wrap up the project and discuss future directions. Sections VI and VII contain the references and source code, respectively.

II. Problem formulation:

We first define our **problem statement**: based on the relative frequencies of the 64 codons in a nucleic acid molecule, we predict the taxonomic domain (archaea, bacteria, eukarya, or virus) of the organism from which the nucleic acid originates. Given this purpose, we formulate our problem as a **classification** problem (**supervised learning**).

The dataset was used in research **by Hallee and Khomtchouk (2023)** [1] and was donated to the **UCI Machine Learning Repository** in 2020 [2]. We obtained the dataset from the UCI repository.

The dataset is relatively large, containing **13,028 data points**. **Each datapoint (observation) in the dataset represents a nucleic acid molecule**, with information regarding its codon usage. There are **69 columns**. The first column is "Kingdom", which contains **categorical** data about the kingdom of the organism to which the nucleic acid belongs, consisting of the following categories: 'arc' (archaea), 'bct' (bacteria), 'phg' (bacteriophage), 'plm' (plasmid), 'pln' (plant), 'inv' (invertebrate), 'vrt' (vertebrate), 'mam' (mammal), 'rod' (rodent), 'pri' (primate), and 'vrl'(virus). Column 2 to column 5 contain some identifiers. Column 6 to column 69 (labelled "UUU", "UUA", "UUF", etc.) contains the relative frequencies of 64 codons, which are **continuous** and recorded as **floats**. These will be our **features**.

Our purpose is to determine the domain of a nucleic acid's "host" organism, but this information is not readily available in the dataset. Hence, we must utilize the "Kingdom" information to infer the corresponding domain. According to the three-domain system, we assign kingdoms 'pln' (plant), 'inv' (invertebrate), 'vrt' (vertebrate), 'mam' (mammal), 'rod' (rodent), and 'pri' (primate) to domain eukarya (labelled 0), kingdom "bct"(bacteria) to domain bacteria (labelled 1), kingdom "arc"(archaea) to domain archaea (labelled 2). Technically, viruses and bacteriophages (bacteria's viruses) are non-living and therefore aren't assigned to any domain in the three-domain system, so we assign both of them to the fourth "virus domain" (labelled 3). The "kingdom" of plasmid ("plm") is an unclear case

– they are assigned to an independent "kingdom" in the dataset, but technically they are additional pieces of nucleic acids inside bacteria. Therefore, to keep things simple, we would exclude plasmids from our analysis. The resulting domain information will be our **categorical labels.** This is the count plot of our labels (**Figure 1**).

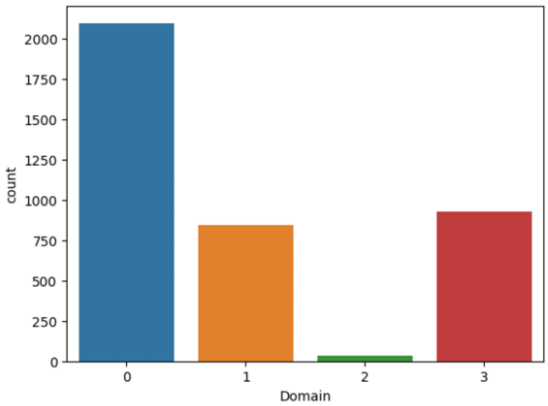


Figure 1

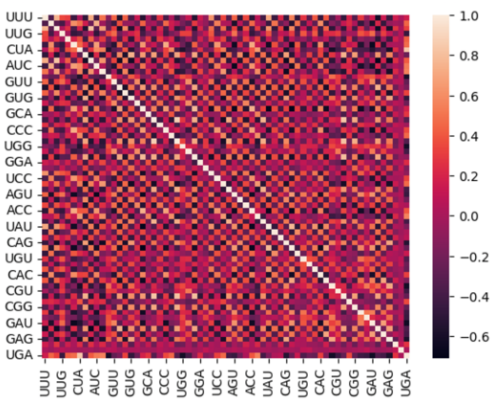


Figure 2

III. Methods:

1. Preprocessing:

As discussed above, we need to label each observation with the appropriate taxonomic domain based on its "Kingdom" feature. A new column, "Domain" is added, containing 4 categories: 0 (eukarya), 1 (bacteria), 2 (archaea), 3 (virus).

Data cleaning: by investigating the data type of each column, we've discovered that among 64 codon frequencies columns, only "UUU" and "UUC" have the dtype object. When trying to cast the entries in these columns to float, we've encountered the strings "non-B hepatitis virus", "12;I", and "-". These are perhaps errors in data entry. Since we have no way to deduce the true values, and there are only 3 erroneous observations, we've decided to drop them from the dataset.

2. Feature selection:

Given our purpose, columns 2 – 5 containing species ID, species name, number of codons in the nucleic acid molecule, and DNA type are irrelevant and can be dropped. Column "Kingdom" must also be dropped after being used to derive labels for our observations. We're left with columns 6 – 69 containing the relative frequencies of 64 codons, which will serve as features for our model.

Can we reduce the number of features? Obviously yes – we can apply some methods such as L1-based feature selection or univariate feature selection to select the "best" features. However, the question is whether we should do so, for our choice of specific codons should have some biological meaning to it. In this regard, there's no biological evidence suggesting that some codons are better than others at predicting the taxonomic domain of an organism. Hence, we've decided to use all 64 codons for our model. Furthermore, even though the dataset is quite large, model training (random forest classifier) is quite fast, so further the dimensionality reduction is not needed from a computational perspective.

3. Feature engineering:

We can see from the count plot of labels (Figure 1) that our dataset is highly imbalanced, with the proportion of label 0 (eukarya) significantly higher than 3 other labels, especially label 2 (archaea). Therefore, we'd use the Synthetic Minority Oversampling Technique (SMOTE) to oversample the minority classes and create a more balanced training data. The way SMOTE works is that it synthesizes

new examples for the minority classes [3]. We'll use the SMOTE implementation of the imblearn Python library.

Scaling the data is not needed, since the data consists of the relative frequencies of codons, so each entry is already between 0 and 1. Furthermore, scaling the columns across observations may distort the relationship between columns (codons), as the proportions of codons in each observation might be changed.

We can look at the heatmap of the pairwise correlation between our features (**Figure 2**). From the heatmap, we can see that collinearity is generally not an issue.

4. Models:

We choose the **Random Forest Classifier (RFC)** as our first model. The reason for this choice is that the random forest algorithm is versatile, stable, and works well for a diverse range of different classification problems. In fact, it's one of the go-to algorithms when it comes to classification. Furthermore, the random forest algorithm is suitable for our relatively high-dimensional dataset, as each decision tree in the forest may only consider a subset of features, reducing the possibility of over-fitting. There's a relatively efficient implementation of the random forest classifier in scikit-learn, which can be readily applied in our project.

For our second model, we select the **Linear Support Vector Classifier (Linear SVC**), which is based on the Support Vector Machine (SVM) algorithm with a linear kernel. The reason for this choice is that Linear SVCs scale well to large datasets and are particularly effective in high-dimensional spaces [5], which is the case for our data. Furthermore, our library of choice, sklearn, also has an implementation of SVC, which makes applying the model relatively straightforward.

5. Loss function:

There are a variety of loss functions for a RFC. Given that we're using a scikit-learn's implementation, it'd be most convenient if we also use one of its readily available loss functions, which include Gini impurity and Entropy. In literature, there has been a theoretical comparison suggesting there's very little different between these 2 loss functions [4]. Gini impurity is a little more efficient, as it doesn't involve computing logarithms. Therefore, we'll go with **Gini impurity**, also scikit-learn's default.

The standard loss for the SVM algorithm is hinge loss. As we're using sklearn's implementation of Linear SVC, there are 2 possible choices for the loss function, hinge loss and squared hinge loss, which is simply the square of hinge loss. Squared hinge loss penalizes large errors more strongly compared to hinge loss and can lead to a smoother loss surface, but otherwise they serve similar purposes. In our case, we'll stick with sklearn's default, which is **squared hinge loss**, as we don't have any particular reason to deviate from it.

6. Evaluation metrics:

Since this is a classification problem, a straightforward evaluation metric would be a model's accuracy. However, since our dataset is highly imbalanced, we'll also consider the macro-averaged F1 score. The reason is that F1 score takes into account both the precision and recall of classes, so a model heavily biased towards the majority class can still have a high accuracy, but it will most likely have a low F1 score. Macro-averaging is most suitable as it assigns equal weight to each class's F1 score, in contrast to weighted or micro-averaging.

7. Training set, validation set, and test set construction:

Given that our dataset is highly imbalanced, we'll use stratified splits to make sure that we maintain the same class distribution in each data subset as in the original dataset. First, we use scikit-learn's train-test-split function with a specified stratify parameter and test size 0.3, resulting in the training + validation set of size 9105 and test set of size 3903. We then perform stratified 5-fold cross validation on this training + validation set, further splitting this set into a smaller training set of size 7284 and a

validation set of size 1821 at each fold. After performing SMOTE, the size of the training set (at each fold) increases to 15480.

IV. Result

Over 5-fold cross validation, our RFC results in an average accuracy of 0.94 (standard deviation 0.004) and an average macro-F1 of 0.89 (standard deviation 0.017). Meanwhile, our Linear SVC results in an average accuracy of 0.82 (standard deviation 0.011) and an average macro-F1 of 0.73 (standard deviation 0.014). It's obvious that RFC performs much better over 5-fold cross validation compared to Linear SVC. The reason for this might be that a linear model is too simple to capture the true relationship between features and labels in our data. Therefore, our final model will be RFC. The RFC is tested with our test set. For this, we train an RFC with our whole training + validation set, resampled with SMOTE, which consists of 9105 data points before SMOTE and increases to 19352 data points after SMOTE. The accuracy of our RFC on the test set is 0.94, and the macro-averaged F1 score is 0.91, which are similar to our validation metrics. This indicates that our training, validation, and test sets are representative of the whole dataset. The classification report is shown in figure 3.

	precision	recall	f1-score	support
0	0.97	0.96	0.96	2074
1	0.93	0.94	0.93	876
2	0.84	0.82	0.83	38
3	0.91	0.91	0.91	915
accuracy			0.94	3903
macro avg	0.91	0.91	0.91	3903
weighted avg	0.94	0.94	0.94	3903

Figure 3: Classification report of the final RFC's performance on the test set

V. Conclusion

In this machine learning project, we've applied a Random Forest Classifier and a Linear Support Vector Classifier with SMOTE resampling to classify organisms into four taxonomic domains based on the relative codon frequencies of their nucleic acids. Our final RFC achieved satisfactory results, yielding an accuracy of 0.94 and a macro-averaged F1 score of 0.91 on the test dataset. As this is a highly imbalanced dataset, a macro-averaged F1 score of 0.91 (as well as the classification report in Figure 3) demonstrates that our model has done a decent job at classifying all 4 classes with satisfactory precision and recall. Our result demonstrates that we can accurately determine the taxonomic domain of an organism with the codon frequencies of its nucleic acids alone, without the need for complicated sequence analysis methods.

Our future steps could be trying to fine-tune the hyperparameters of the RFC model to obtain better results, as well as exploring other machine learning methods such as gradient boosting, support vector machine with non-linear kernels, and artificial neural networks. We could also try to classify organisms at finer taxonomic levels such as kingdom or phylum or try to determine the cellular compartment of the nucleic acid molecule.

VI. Reference

[1] Hallee, L. and Khomtchouk, B.B. (2023) 'Machine learning classifiers predict key genomic and evolutionary traits across the kingdoms of life', Scientific Reports, 13(1), p. 2088. doi:10.1038/s41598-023-28965-7.

[2] L. Hallee, B.B.K. (2020) 'Codon usage'. doi:10.24432/C5KP6B.

[3] Chawla, N.V. et al. (2011) 'Smote: synthetic minority over-sampling technique'. doi:10.48550/ARXIV.1106.1813.

[4] Raileanu, L.E. and Stoffel, K. (2004) 'Theoretical comparison between the gini index and information gain criteria', Annals of Mathematics and Artificial Intelligence, 41(1), pp. 77–93. doi:10.1023/B:AMAI.0000018580.96245.c6.

[5] https://scikit-learn.org/stable/modules/svm.html

VII. Appendix: Source code

Predicting an organism's taxonomic domain based on its nucleic's acid codon usage

We first import necessary packages and read our data.

In [ ]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import classification_report, f1_score, accuracy_score
from sklearn.model_selection import train_test_split, RandomizedSearchCV, StratifiedKFold
from tqdm import tqdm
from sklearn.ensemble import RandomForestClassifier
from sklearn.svm import LinearSVC, SVC
from sklearn.neighbors import KNeighborsClassifier
from imblearn.over_sampling import SMOTE
import seaborn as sns
from sklearn.svm import SVC
```

In [ ]:
```
data = pd.read_csv("codon_usage.csv")
data.head()
```

```
/var/folders/yy/qhmg6fq12dj6f32h2788192c0000gn/T/ipykernel_44865/3156801676.py:1: DtypeWarning: Columns (5,6) hav
e mixed types. Specify dtype option on import or set low_memory=False.
  data = pd.read_csv("codon_usage.csv")
```

Out[ ]:

	Kingdom	DNAtype	SpeciesID	Ncodons	SpeciesName	UUU	UUC	UUA	UUG	CUU	...	CGG	AGA	AGG	GAU
0	vrl	0	100217	1995	Epizootic haematopoietic necrosis virus	0.01654	0.01203	0.00050	0.00351	0.01203	...	0.00451	0.01303	0.03559	0.01003
1	vrl	0	100220	1474	Bohle iridovirus	0.02714	0.01357	0.00068	0.00678	0.00407	...	0.00136	0.01696	0.03596	0.01221
2	vrl	0	100755	4862	Sweet potato leaf curl virus	0.01974	0.0218	0.01357	0.01543	0.00782	...	0.00596	0.01974	0.02489	0.03126
3	vrl	0	100880	1915	Northern cereal mosaic virus	0.01775	0.02245	0.01619	0.00992	0.01567	...	0.00366	0.01410	0.01671	0.03760
4	vrl	0	100887	22831	Soil-borne cereal mosaic virus	0.02816	0.01371	0.00767	0.03679	0.01380	...	0.00604	0.01494	0.01734	0.04148

5 rows × 69 columns

In [ ]:
```
len(data)
```

Out[ ]:
```
13028
```

Let's now label our observations with the appropriate domain.

- Kingdoms 'pln' (plant), 'inv' (invertebrate), 'vrt' (vertebrate), 'mam' (mammal), 'rod' (rodent), and 'pri' (primate): domain eukarya - label 0
- Kingdom 'bct'(bacteria): domain bacteria - label 1
- Kingdom 'arc'(archaea): domain archaea - label 2
- Kingdom 'vrl'(virus) and 'phg'(bacteriophage): "domain" virus - label 3

Observations with Kingdom = 'plm'(plasmid) are dropped.

In [ ]:
```
virus = ["phg", "vrl"]
data = data[data["Kingdom"] != "plm"]
data["Domain"] = data["Kingdom"].apply(lambda k: 1 if k == "bct" else 2 if k == "arc" else 3 if k in virus else 0
data.head()
```

Out[ ]:

	Kingdom	DNAtype	SpeciesID	Ncodons	SpeciesName	UUU	UUC	UUA	UUG	CUU	...	AGA	AGG	GAU	GAC
0	vrl	0	100217	1995	Epizootic haematopoietic necrosis virus	0.01654	0.01203	0.00050	0.00351	0.01203	...	0.01303	0.03559	0.01003	0.04612
1	vrl	0	100220	1474	Bohle iridovirus	0.02714	0.01357	0.00068	0.00678	0.00407	...	0.01696	0.03596	0.01221	0.04545
2	vrl	0	100755	4862	Sweet potato leaf curl virus	0.01974	0.0218	0.01357	0.01543	0.00782	...	0.01974	0.02489	0.03126	0.02036

3	vrl	0	100880	1915	Northern cereal mosaic virus	0.01775	0.02245	0.01619	0.00992	0.01567	...	0.01410	0.01671	0.03760	0.01932
4	vrl	0	100887	22831	Soil-borne cereal mosaic virus	0.02816	0.01371	0.00767	0.03679	0.01380	...	0.01494	0.01734	0.04148	0.02483

5 rows × 70 columns

Data cleaning

Let's call the DataFrame's info() method to get an overview of our dataset, and check for null entries and duplicated observations.

In [ ]:
```
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 13010 entries, 0 to 13027
Data columns (total 70 columns):
 #   Column       Non-Null Count  Dtype
---  ------       --------------  -----
 0   Kingdom      13010 non-null  object
 1   DNAtype      13010 non-null  int64
 2   SpeciesID    13010 non-null  int64
 3   Ncodons      13010 non-null  int64
 4   SpeciesName  13010 non-null  object
 5   UUU          13010 non-null  object
 6   UUC          13010 non-null  object
 7   UUA          13010 non-null  float64
 8   UUG          13010 non-null  float64
 9   CUU          13010 non-null  float64
 10  CUC          13010 non-null  float64
 11  CUA          13010 non-null  float64
 12  CUG          13010 non-null  float64
 13  AUU          13010 non-null  float64
 14  AUC          13010 non-null  float64
 15  AUA          13010 non-null  float64
 16  AUG          13010 non-null  float64
 17  GUU          13010 non-null  float64
 18  GUC          13010 non-null  float64
 19  GUA          13010 non-null  float64
 20  GUG          13010 non-null  float64
 21  GCU          13010 non-null  float64
 22  GCC          13010 non-null  float64
 23  GCA          13010 non-null  float64
 24  GCG          13010 non-null  float64
 25  CCU          13010 non-null  float64
 26  CCC          13010 non-null  float64
 27  CCA          13010 non-null  float64
 28  CCG          13010 non-null  float64
 29  UGG          13010 non-null  float64
 30  GGU          13010 non-null  float64
 31  GGC          13010 non-null  float64
 32  GGA          13010 non-null  float64
 33  GGG          13010 non-null  float64
 34  UCU          13010 non-null  float64
 35  UCC          13010 non-null  float64
 36  UCA          13010 non-null  float64
 37  UCG          13010 non-null  float64
 38  AGU          13010 non-null  float64
 39  AGC          13010 non-null  float64
 40  ACU          13010 non-null  float64
 41  ACC          13010 non-null  float64
 42  ACA          13010 non-null  float64
 43  ACG          13010 non-null  float64
 44  UAU          13010 non-null  float64
 45  UAC          13010 non-null  float64
 46  CAA          13010 non-null  float64
 47  CAG          13010 non-null  float64
 48  AAU          13010 non-null  float64
 49  AAC          13010 non-null  float64
 50  UGU          13010 non-null  float64
 51  UGC          13010 non-null  float64
 52  CAU          13010 non-null  float64
 53  CAC          13010 non-null  float64
 54  AAA          13010 non-null  float64
 55  AAG          13010 non-null  float64
 56  CGU          13010 non-null  float64
 57  CGC          13010 non-null  float64
 58  CGA          13010 non-null  float64
 59  CGG          13010 non-null  float64
 60  AGA          13010 non-null  float64
 61  AGG          13010 non-null  float64
```

```
 62  GAU          13010 non-null  float64
 63  GAC          13010 non-null  float64
 64  GAA          13010 non-null  float64
 65  GAG          13010 non-null  float64
 66  UAA          13010 non-null  float64
 67  UAG          13010 non-null  float64
 68  UGA          13010 non-null  float64
 69  Domain       13010 non-null  int64
dtypes: float64(62), int64(4), object(4)
memory usage: 7.0+ MB
```

In [ ]:
```
data.isnull().sum().sum(), data.duplicated().sum()
```

Out[ ]:
```
(0, 0)
```

We can see that there isn't any null entry and duplicated observations.

Furthermore, the datatype of the relative frequencies of all codons are float64, except for UUU and UUC whose datatype is object. We'll try to convert them to float64.

In [ ]:
```
def convert_to_float(x):
    try:
        return float(x)
    except ValueError:
        print(x)
        return None
```

In [ ]:
```
data["UUU"].apply(convert_to_float)
```

Out[ ]:
```
non-B hepatitis virus
12;I
0        0.01654
1        0.02714
2        0.01974
3        0.01775
4        0.02816
          ...
13023    0.02552
13024    0.01258
13025    0.01423
13026    0.01757
13027    0.01778
Name: UUU, Length: 13010, dtype: float64
```

We can see that in the "UUU" column, there are 2 entries that can't be converted into float.

In [ ]:
```
data["UUC"].apply(convert_to_float)
```

Out[ ]:
```
-
0        0.01203
1        0.01357
2        0.02180
3        0.02245
4        0.01371
          ...
13023    0.03555
13024    0.03193
13025    0.03321
13026    0.02028
13027    0.03724
Name: UUC, Length: 13010, dtype: float64
```

In the "UUC column, there is 1 entry that can't be converted into float. We'll change the dataframe in-place and drop these erroneous entries.

In [ ]:
```
data["UUU"] = data["UUU"].apply(convert_to_float)
data["UUC"] = data["UUC"].apply(convert_to_float)
data.dropna(inplace=True)
```

```
non-B hepatitis virus
12;I
-
```

To keep our dataframe clean, let's also drop columns unnecessary for our analysis.

In [ ]:

```python
cols_to_drop = ["Kingdom", "DNAtype", "SpeciesID", "Ncodons", "SpeciesName"]
data.drop(columns=cols_to_drop, inplace=True)
```

In [ ]:

```python
data.shape
```

Out[ ]:

```
(13008, 65)
```

Our cleaned dataframe now contains 13008 rows and 65 columns, among which 64 are features and the remaining one is the label.

Exploratory data analysis

We first take a look of class distribution.

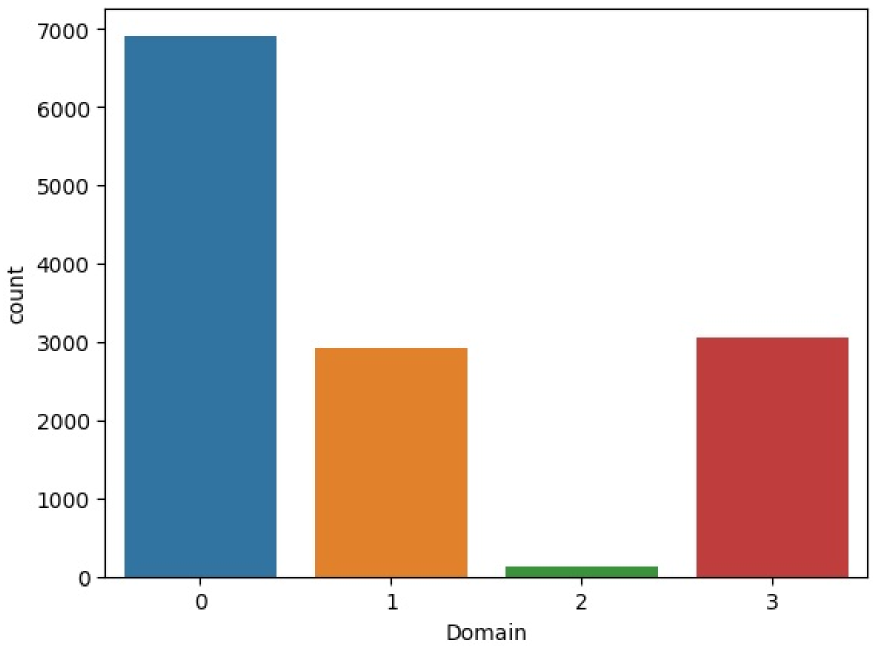
In [ ]:

```python
sns.countplot(x=data["Domain"])
```

```
/Users/macbookair/ml-prj/venv/lib/python3.9/site-packages/seaborn/_oldcore.py:1498: FutureWarning: is_categorical
_dtype is deprecated and will be removed in a future version. Use isinstance(dtype, CategoricalDtype) instead
  if pd.api.types.is_categorical_dtype(vector):
/Users/macbookair/ml-prj/venv/lib/python3.9/site-packages/seaborn/_oldcore.py:1498: FutureWarning: is_categorical
_dtype is deprecated and will be removed in a future version. Use isinstance(dtype, CategoricalDtype) instead
  if pd.api.types.is_categorical_dtype(vector):
/Users/macbookair/ml-prj/venv/lib/python3.9/site-packages/seaborn/_oldcore.py:1498: FutureWarning: is_categorical
_dtype is deprecated and will be removed in a future version. Use isinstance(dtype, CategoricalDtype) instead
  if pd.api.types.is_categorical_dtype(vector):
```

Out[ ]:

```
<Axes: xlabel='Domain', ylabel='count'>
```



Our dataset is highly imbalanced.

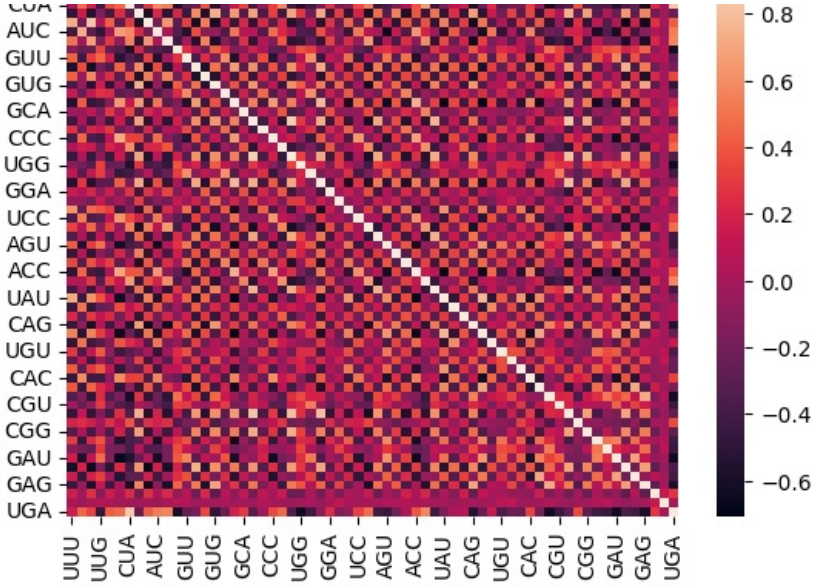
Now, let's visulize the pairwise correlation between our features.

In [ ]:

```python
sns.heatmap(data.iloc[:, :-1].corr())
```

Out[ ]:

```
<Axes: >
```



Preparing training and testing datasets

We use the sklearn's train_test_split(). Since our data is highly imbalanced, we specify the stratify parameter as the labels to preserve the class distribution in both the train and test sets.

In [ ]:
```
X_train_val, X_test, y_train_val, y_test = train_test_split(data.iloc[:, :-1].to_numpy(), data.iloc[:, -1].to_num
X_train_val.shape, X_test.shape, y_train_val.shape, y_test.shape
```

Out[ ]:
```
((9105, 64), (3903, 64), (9105,), (3903,))
```

5-fold cross validation

We now perform stratified 5-fold cross validation on our data with an untuned RandomForestClassifier (default hyperparameters) from sklearn. For each fold, we perform SMOTE on the training set, train the model on the SMOTE-resampled training set, and compute the accuracy and the macro-averaged F1 score of the model on the validation set.

In [ ]:
```
skf = StratifiedKFold(n_splits=5) # stratified 5-fold
accs = [] # array to save each fold's accuracy
f1_macros = [] # array to save each fold's macro-averaged f1
for i, (train_index, val_index) in enumerate(skf.split(X_train_val, y_train_val)):
    # divide into train and val sets
    train_X = X_train_val[train_index]
    train_y = y_train_val[train_index]
    val_X = X_train_val[val_index]
    val_y = y_train_val[val_index]
    # perform SMOTE on the train set
    sm = SMOTE()
    train_X_sm, train_y_sm = sm.fit_resample(train_X, train_y)
    # initiate and train the model
    rfc = RandomForestClassifier()
    rfc.fit(train_X_sm, train_y_sm)
    # predict the labels of val set
    preds = rfc.predict(val_X)
    # compute macro-averaged f1 score and accuracy score
    f1 = f1_score(preds, val_y, average="macro")
    acc = accuracy_score(preds, val_y)
    f1_macros.append(f1)
    accs.append(acc)
    print(f"Fold {i}: train size {len(train_X)}, val size {len(val_X)}, train size SMOTE {len(train_X_sm)}, accu
```

```
Fold 0: train size 7284, val size 1821, train size SMOTE 15480, accuracy 0.943986820428336, f1 0.86611026455661
Fold 1: train size 7284, val size 1821, train size SMOTE 15480, accuracy 0.9395936298736958, f1 0.904483968509943
6
Fold 2: train size 7284, val size 1821, train size SMOTE 15484, accuracy 0.9373970345963756, f1 0.881300983086071
3
Fold 3: train size 7284, val size 1821, train size SMOTE 15484, accuracy 0.9494783086216365, f1 0.915905824905536
3
Fold 4: train size 7284, val size 1821, train size SMOTE 15480, accuracy 0.9450851180669961, f1 0.893700726471021
7
```

We can calculate the mean and standard deviation of the accuracy and f1 score across different folds.

In [ ]:
```
np.mean(accs), np.std(accs)
```

Out[ ]:
```
(0.9431081823174081, 0.004248013092984913)
```

In [ ]:
```
np.mean(f1_macros), np.std(f1_macros)
```

Out[ ]:
```
(0.8923003535058365, 0.01740417598056037)
```

Our model obtained an accuracy of 0.94 and a macro-averaged F1 score of 0.89 across 5-fold cross validation, which is satisfactory.

Let's compare this result with the approach that doesn't use SMOTE for class rebalancing.

In [ ]:
```
accs = [] # array to save each fold's accuracy
f1_macros = [] # array to save each fold's macro-averaged f1
for i, (train_index, val_index) in enumerate(skf.split(X_train_val, y_train_val)):
    # divide into train and val sets
    train_X = X_train_val[train_index]
    train_y = y_train_val[train_index]
    val_X = X_train_val[val_index]
    val_y = y_train_val[val_index]
    # initiate and train the model
    rfc = RandomForestClassifier()
    rfc.fit(train_X, train_y)
    # predict the labels of val set
    preds = rfc.predict(val_X)
    # compute macro-averaged f1 score and accuracy score
    f1 = f1_score(preds, val_y, average="macro")
    acc = accuracy_score(preds, val_y)
    f1_macros.append(f1)
    accs.append(acc)
    print(f"Fold {i}: train size {len(train_X)}, val size {len(val_X)}, accuracy {acc}, f1 {f1}")
```

```
Fold 0: train size 7284, val size 1821, accuracy 0.9308072487644151, f1 0.7475396272871563
Fold 1: train size 7284, val size 1821, accuracy 0.9335529928610653, f1 0.8563552255578795
Fold 2: train size 7284, val size 1821, accuracy 0.9401427786930258, f1 0.7748352126601699
Fold 3: train size 7284, val size 1821, accuracy 0.9467325645249862, f1 0.8339201571781971
Fold 4: train size 7284, val size 1821, accuracy 0.943986820428336, f1 0.8533682895816143
```

In [ ]:
```
np.mean(accs), np.std(accs)
```

Out[ ]:
```
(0.9390444810543658, 0.006045627213409283)
```

In [ ]:
```
np.mean(f1_macros), np.std(f1_macros)
```

Out[ ]:
```
(0.8132037024530033, 0.04401893065087388)
```

We can see that SMOTE improves our macro-averaged F1 score.

Now, let's perform 5-fold cross validation with a Linear SVC. We'll also implement SMOTE on the training set at each fold.

In [ ]:
```
accs = [] # array to save each fold's accuracy
f1_macros = [] # array to save each fold's macro-averaged f1
for i, (train_index, val_index) in enumerate(skf.split(X_train_val, y_train_val)):
    # divide into train and val sets
    train_X = X_train_val[train_index]
    train_y = y_train_val[train_index]
    val_X = X_train_val[val_index]
    val_y = y_train_val[val_index]
    # perform SMOTE on the train set
    sm = SMOTE()
    train_X_sm, train_y_sm = sm.fit_resample(train_X, train_y)
    # initiate and train the model
    svc = LinearSVC(dual="auto")
    svc.fit(train_X_sm, train_y_sm)
    # predict the labels of val set
```

```
    preds = svc.predict(val_X)
    # compute macro-averaged f1 score and accuracy score
    f1 = f1_score(preds, val_y, average="macro")
    acc = accuracy_score(preds, val_y)
    f1_macros.append(f1)
    accs.append(acc)
    print(f"Fold {i}: train size {len(train_X)}, val size {len(val_X)}, train size SMOTE {len(train_X_sm)}, accu
```

```
Fold 0: train size 7284, val size 1821, train size SMOTE 15480, accuracy 0.8204283360790774, f1 0.724319006312097
Fold 1: train size 7284, val size 1821, train size SMOTE 15480, accuracy 0.8215266337177375, f1 0.718732218408568
4
Fold 2: train size 7284, val size 1821, train size SMOTE 15484, accuracy 0.8017572762218561, f1 0.706286012658369
7
Fold 3: train size 7284, val size 1821, train size SMOTE 15484, accuracy 0.8314113124656782, f1 0.746048464317659
7
Fold 4: train size 7284, val size 1821, train size SMOTE 15480, accuracy 0.8303130148270181, f1 0.738669579354719
9
```

In [ ]:
```
np.mean(accs), np.std(accs)
```

Out[ ]:
```
(0.8210873146622735, 0.01063932436095539)
```

In [ ]:
```
np.mean(f1_macros), np.std(f1_macros)
```

Out[ ]:
```
(0.726811056210283, 0.014166564774752626)
```

We can see that RFC gives much better result compared to Linear SVC over 5-fold cross validation. Therefore, RFC will be our final model.

Testing

We now evaluate our chosen model using the test set.

In [ ]:
```
len(X_train_val)
```

Out[ ]:
```
9105
```

In [ ]:
```
sm = SMOTE()
X_train_val_sm, y_train_val_sm = sm.fit_resample(X_train_val, y_train_val)
len(X_train_val_sm)
```

Out[ ]:
```
19352
```

In [ ]:
```
rfc = RandomForestClassifier()
rfc.fit(X_train_val_sm, y_train_val_sm)
y_preds = rfc.predict(X_test)
print(classification_report(y_test, y_preds))
```

```
              precision    recall  f1-score   support

           0       0.97      0.96      0.96      2074
           1       0.93      0.94      0.93       876
           2       0.84      0.82      0.83        38
           3       0.91      0.91      0.91       915

    accuracy                           0.94      3903
   macro avg       0.91      0.91      0.91      3903
weighted avg       0.94      0.94      0.94      3903
```

As seen in the classification report, our RFC has done a decent job at classifying all 4 classes!