Task 2: Ensemble Classifiers

Roger Llorenç, Víctor Villegas, Luis Sierra (Group 7)

May 20, 2024

Contents

1 Introduction

1.1 Objectives

Our main objective with this work is that of building and fitting ensemble-based tree models that, based on a dataset of cancer patients, predicts recurrence (reappearance of the disease) of cancer from putative recurrence biomarkers. To do so, we build, tune and compare (using `python`) three different types of models - Random Forests, Gradient Boosting and XGBoost, and then evaluate their performance on our dataset.

1.2 Materials and methods

1.2.1 Datasets and packages

The main python libraries used across the code are: - `numpy`: fundamental package for scientific computing in Python. - `pandas`: for a more powerful handling of datasets. - `pyplot`: we will use this submodule from `matplotlib` to do most of the plots. - `sklearn`: open source tool built on top of NumPy, and matplotlib. Along with its submodules, it will be the package used to carry out the coding related to the learning procedures. The submodules imported include tools to tune the models (such as `GridSearchCV`) or to build specific models (e.g. `GradientBoostingClassifier`)

Apart from the libraries pre-imported in the following cell, we import other `sklearn` submodules when needed and the libraries: `itertools` (for readable and fast iteration) and `xgboost` (to build the model of the same name).

The datasets used are found in the `cancerDat.csv` and `cancerInfo.csv`, and include of cancer patients with the presence of 102 different peptides in the patients' organisms, which have been found to be related to recurrence. These are broken down and studied in the following section.

1.2.2 Procedure

The procedure followed involves four main steps.

Firstly, we clean up the data to ensure it is ready to be used to build the model. This step involves the handling of missing data, renaming the variables and encoding them correctly. To that effect we perform a simple exploratory data analysis.

Secondly, we randomly partition our data into train and test splits using functions from the `scikit-learn` library. The split will be of size 2/3 for training and 1/3 for testing.

Thirdly, we will build the models, a Random Forest Classifier, a Gradient Boosted decision tree and XGBoost, on the training data (with the imported libraries) and tune their hyperparameters via a grid search.

Lastly, we will analyse the strength of our models by looking at their predicted performance using the test set and evaluate their strengths and weaknesses.

This study has been developed first answering the Questions posed in the assignment one by one and then developing a cohesive report exposing and analysing the results in depth. We have chosen Python to develop the program out of personal preference, but the R programming language is also a viable option.

2 Exploratory Data Analysis

We visualize the structure of the data after importing the csv format files. We can see how the data is composed of a table of 129 patients with 102 variables, 101 of which are peptide biomarkers and the the other one is the response variable on the recurrence.

The other table corresponds to the same patients but with a single, extra variable `site`, which is of factor type, that may be related to the kind of recurrence, but the authors are not experts in the domain of clinical oncology. As such, it was decided to omit the treatment of this variable to focus solely on the statistical analysis of the raw recurrence.

[6]:
```
print(df1.shape)
print(df2.shape)
```

```
(129, 102)
(129, 1)
```

By a coarse inspection of the datafile, we can immediately see that the data ingestion pipeline will require some work, given the combination of the response and patient index into the same variable.

First, we define a single response variable according to whether the cancer sees a recurrence or not. This is originally coded into the key (index) of the patient, so a separate binary vector is created to encode this response, where `0` denotes non-recurrence and `1` denotes recurrence, stored as a Pandas series `y = labels_1`.

[10]:
```
y = pd.Series(y)
```

The next step in the exploration phase is checking for missing values and calculating some descriptive statistics of the data. With the Pandas library one can easily locate the missing values; in fact there is a substantial amount of missing values (5151) in the dataset, so in order to avoid discarding such a large amount of information, we resort to imputing the linear interpolation of the missing entries, as calculated by Pandas' `interpolate` method. This artificially increases the amount of information that we work with, but we claim that for the final results this is preferable to eliminating the entire rows and rendering our dataset significantly smaller. The following outputs show the amount of missing values and how they are removed by linear interpolation.

[20]:
```
for col in df1:
    df1[col] = pd.to_numeric(df1[col], errors='coerce')
df1.dtypes;
```

[21]:
```
print(df1.isnull().values.any())
print(sum(df1.isnull()))
df1 = df1.interpolate(method ='linear', limit_direction ='forward')
print(df1.isnull().values.any())
```

```
True
5151
False
```

Now that we have handled the missing values, we can go ahead and perform some more sophisticated exploration techniques. We provide a small snapshot (the first 9 variables) of the descriptive statistics of the dataset, but note that due to its large size, it is only possible to show a handful of the biomarker variables.

[9]:
```
df1.describe()
```

[9]:

	0	1	2	3	4	5	\
count	129.000000	129.000000	129.000000	129.000000	129.000000	129.000000	
mean	19.930039	23.286214	18.226328	17.331228	22.591467	20.193446	
std	1.646143	0.784218	1.126192	1.477403	1.562914	1.185180	
min	15.836862	21.313713	14.401524	12.976690	18.807566	16.963778	

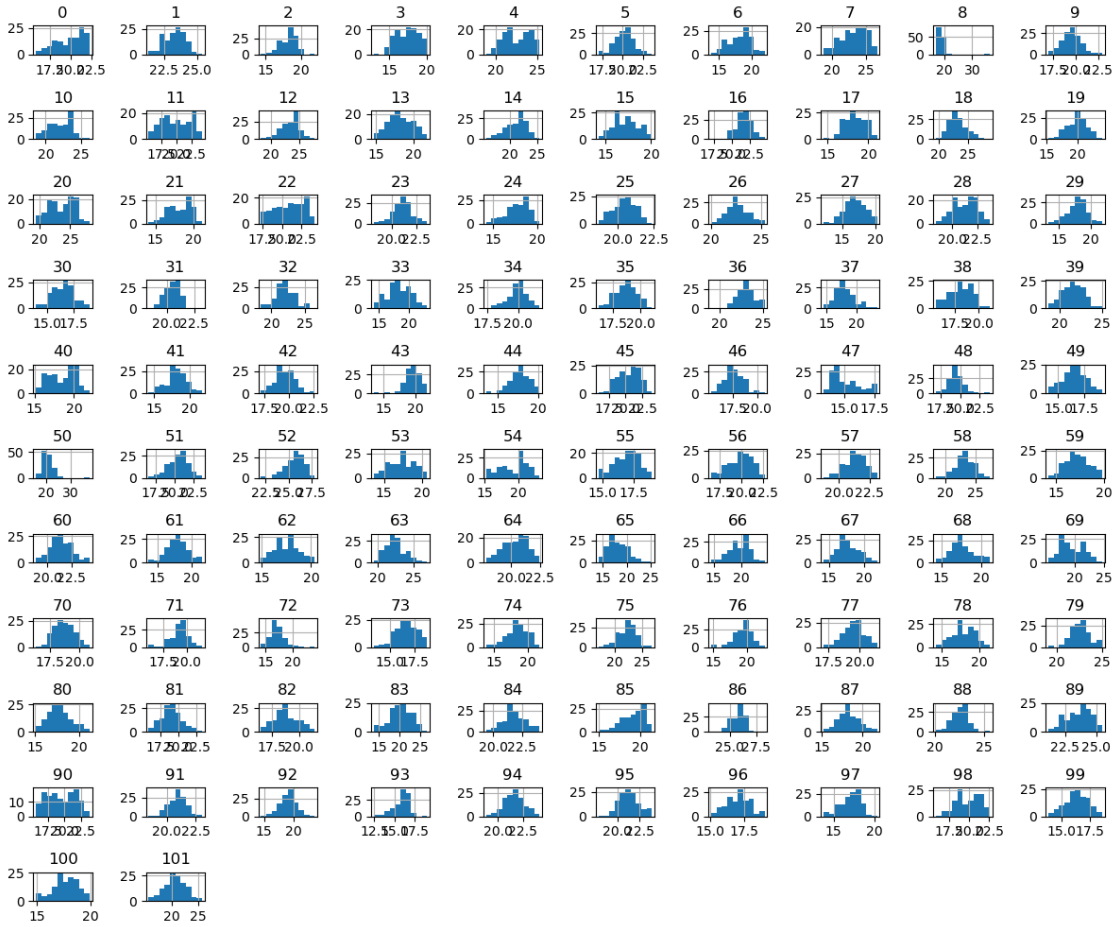
```
25%     18.547029   22.755110   17.440432   16.166380   21.321595   19.397560
50%     20.341514   23.374865   18.349787   17.373647   22.461809   20.264099
75%     21.362745   23.793517   18.968572   18.501081   23.992847   20.961033
max     22.315946   25.387870   21.837865   19.994349   25.306821   23.671558

              6           7           8           9    ...
count   129.000000  129.000000  129.000000  129.000000  ...
mean     18.133973   23.193300   18.687522   19.516615  ...
std       1.751196    1.860021    1.744740    1.068425  ...
min      13.770822   18.824646   16.818849   16.952402  ...
25%      17.001766   22.036426   18.109188   18.830477  ...
50%      18.421874   23.469361   18.548956   19.546749  ...
75%      19.426085   24.633681   18.967436   20.108405  ...
max      21.934433   26.596796   36.306672   22.920624  ...

[8 rows x 102 columns]
```

In the following compound graph, the distribution of each of the 101 biomarkers is shown in histogram form. In this way, one can observe at a birdseye view the distribution of the different peptide markers.

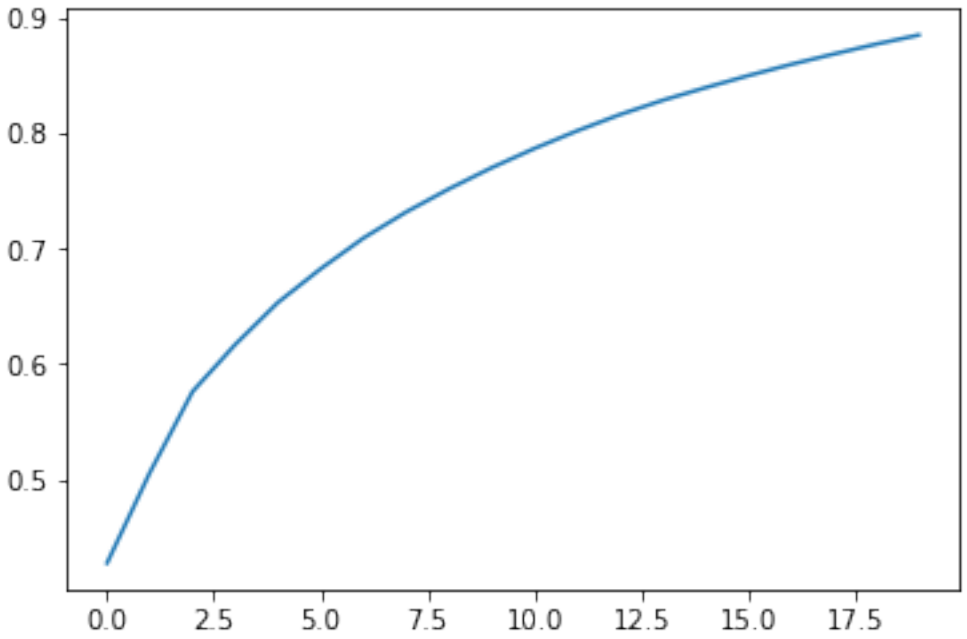
From the histograms, one can observe some erratic distributions for some of the covariates, where they do not seem to fit very well with the statement of the law of large numbers. By this we mean that the shapes of the distributions do not fit our expectations of normal distributions. Specifically, as an example, the covariates indexed in positions `15`, `54` or `78` have some interesting shapes as they show multimodal distributions and signs of heavy tails.



We continue the analysis with a Principal Component Analysis (PCA), which lets us identify the directions of largest variation in the data. The data is of high dimension, given that we have 101 explanatory variables, so using a dimensionality reduction technique allows us to potentially examine a how the datapoints could fit into a lower dimensional subspace. We conclude that the amount of variance accumulated by the first twenty principal components amounts to almost 90%. This justifies our final choice of having the classifiers only include ten peptides as features to train on.

[11]:
```
plt.plot(list(itertools.accumulate(pca.explained_variance_ratio_)))
```

[11]:
```
[<matplotlib.lines.Line2D at 0x779719e76560>]
```



3 Train-Test Split

We separate the data into two sets: the training set, containing two thirds of the data, and the test set, containing the rest. We will use these sets to fit different models in the next section. We also set a numpy seed to allow for the reproducibility of the results. The exact size of the train-test split is provided for completeness.

[26]:
```
np.random.seed(1234)
```

[32]:
```
print(X_train.shape, X_test.shape, y_train.shape, y_test.shape)
```

```
(86, 102) (43, 102) (86,) (43,)
```

4 Model building and fitting

The following models are fitted: a Random Forest Classifier and two Boosting Classifiers (Gradient boosting and XGBoost).

4.1 Random Forest Classifier

A random forest classifier is built using the `scikit-learn` library, with the same seed as the one used in the train-test split and limiting the number of total features to 10. The hyperparameters are explored and tuned using a grid search procedure, which is automated thanks to internal methods also from the `scikit-learn` library.

[24]:
```
from sklearn.ensemble import RandomForestClassifier
rf = RandomForestClassifier(random_state = 1234)
parameters = {'n_estimators' : range(10,130,20),
              'max_features' : (1,2,3,4,5,6,7,8,9,10)
             }
```

A simple implementation of the grid search procedure with a $PMSE$ given by cross validation is achieved by calling the `GridSearchCV` function, shown in the following cell.

[33]:
```
grid_rf = GridSearchCV(rf, parameters)
grid_rf.fit(X_train, y_train)
```

[33]:
```
GridSearchCV(estimator=RandomForestClassifier(random_state=1234),
             param_grid={'max_features': (1, 2, 3, 4, 5, 6, 7, 8, 9, 10),
                         'n_estimators': range(10, 130, 20)})
```

The following table shows the results of the grid search procedure, where the hyperparameters of the best model after validation are shown to be 7 total features and 10 weak estimators. Note that the performance is quite unimpressive, barely improving on the null model.

[34]:

	param_max_features	param_n_estimators	mean_test_score	std_test_score
36	7	10	0.594118	0.118234
55	10	30	0.582353	0.079792
48	9	10	0.572549	0.142693
1	1	30	0.569935	0.044213

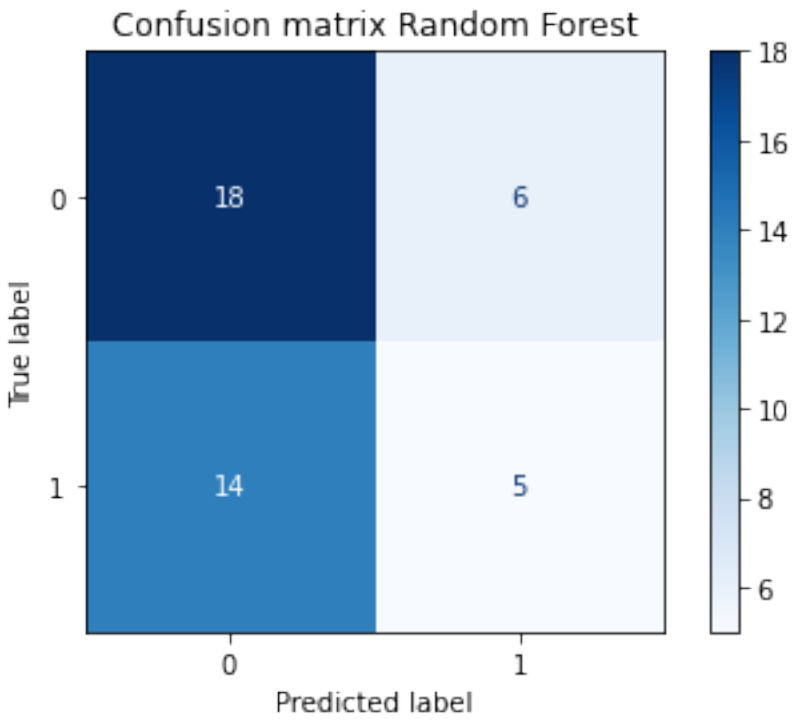
```
Best Parameters: {'max_features': 7, 'n_estimators': 10}
```

[35]:
```
RandomForestClassifier(max_features=7, n_estimators=10, random_state=1234)
```

The accuracy and performance of the Random Forest on the test set is shown below, with a confusion matrix to showcase its quantitiy of type I and type II errors.

[19]:
```
y_pred = best_rf.predict(X_test)
accuracy_score(y_pred, y_test)
```

[19]:
```
0.5348837209302325
```



Finally, the most relevant variables in the prediction, as defined by a weighted average of how many times they appear in a node of the random forest are given by the following table.

	Feature	Importance
51	51	0.057151
11	11	0.043937
25	25	0.032852
96	96	0.030232
78	78	0.029687
76	76	0.028671
5	5	0.027653
34	34	0.026475
12	12	0.026456
62	62	0.024448

4.2 Gradient Boosting

Using stumps as classification trees for the response variable, we now build a gradient boosting model on the stumps as weak learners to then compute the misclassification rates of both the learning set and the test set, across 2,000 iterations. As was done with the random forest in the previous subsection, the hyperparameters of the model are tuned with a grid search.

[36]:
```
from sklearn.ensemble import GradientBoostingClassifier
gb = GradientBoostingClassifier(n_estimators=2000, random_state=1234)
parameters = {'max_depth' : (1,4,8,16)}
```

[37]:
```
grid_gb = GridSearchCV(gb, parameters)
grid_gb.fit(X_train, y_train)
```

[37]:
```
GridSearchCV(estimator=GradientBoostingClassifier(n_estimators=2000,
                                                  random_state=1234),
             param_grid={'max_depth': (1, 4, 8, 16)})
```

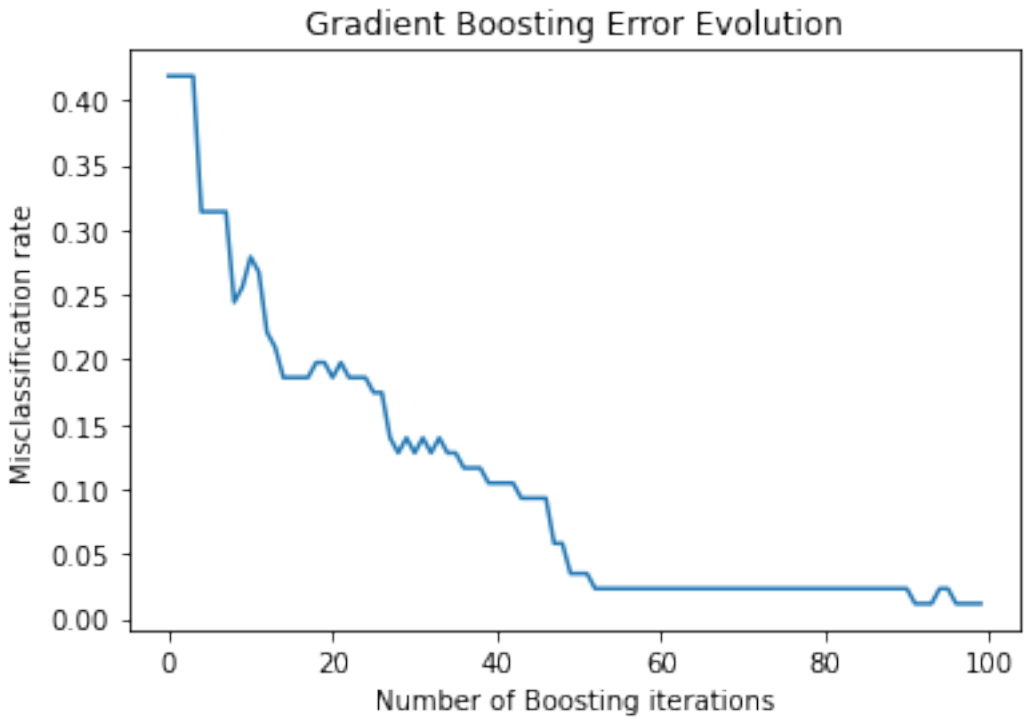
[24]:

	param_max_depth	mean_test_score	std_test_score
0	1	0.524183	0.069834
2	8	0.454248	0.103714
3	16	0.454248	0.103714
1	4	0.442484	0.090319

The results of comparing the validation-set misclassification rates attained by different ensemble classifiers based on trees with varying maximum depth is shown in the previous table, where we find the stumps as being the best performing validation-set learners. We may observe already hints of the strong overfitting tendencies of this type of model, given their poor out-of-sample performance. In fact, we may track the training error across iterations, and as shown in the following plot *Gradient Boosting Error Evolution*, the training error decreases to almost perfection, while the validation set error remains close to the null model, which is a clear sign of overfitting.

```
Best Parameters: {'max_depth': 1}
```

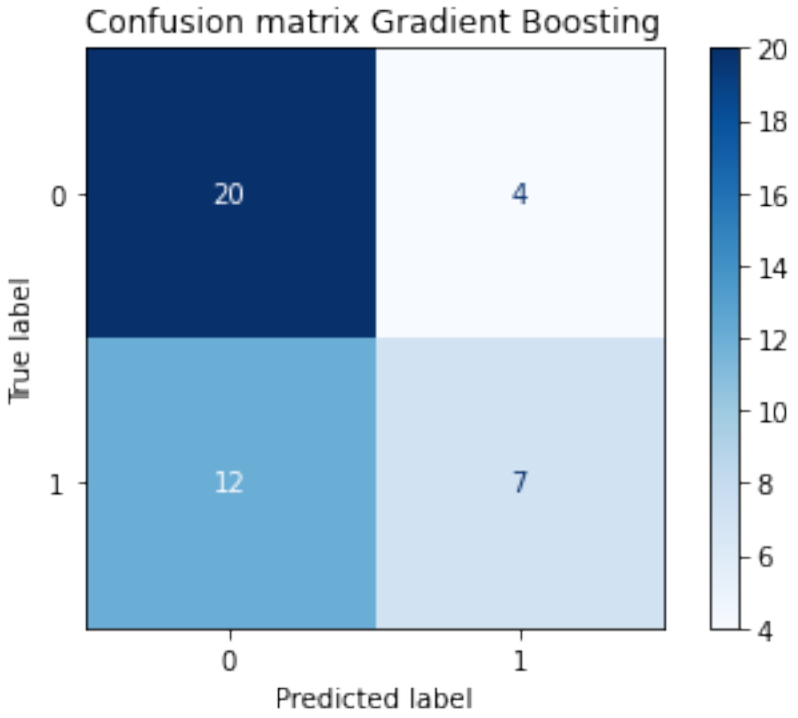
[25]:
```
Text(0.5, 1.0, 'Gradient Boosting Error Evolution')
```



The test-set accuracy and performance of the Gradient Boosting is as follows, with its corresponding confusion matrix in the figure below.

[34]:
```
y_pred = best_gb.predict(X_test)
accuracy_score(y_pred, y_test)
```

[34]:
```
0.627906976744186
```



4.3 XGBoost

To conclude, we introduce another boosting flavour and analyse its performance. It is well-known that the XGBoost attains state of the art performance in a wide variety of learning tasks, although as most models of its kind, it also shows a tendency to overfit the training data. As in the previous two cases, the model's hyperparameters, which are the maximum depth of the decision trees and the learning rate, are explored in a grid search.

[38]:
```
import xgboost
xgb = xgboost.XGBClassifier(n_estimators=2000, random_state=1234)
parameters = {'max_depth' : (1,4,8,16),
              'learning_rate' : [0,2]
              }
```

[42]:
```
grid_xgb = GridSearchCV(xgb, parameters)
grid_xgb.fit(X_train, y_train)
```

The best performing out-of-sample models are shown in the following table, where the best four all have the same error rate (the one corresponding to the null model) and a learning rate of 0. This is because the model strongly overfits the data, which is a consequence of the extremely noisy dataset. In fact, due to the the powerful fitting capabilities of XGBoost, it strongly favours fitting the large noise in the dataset instead of sticking to the weak signal present. Some recent analysis on this behaviour and why zero training error is sometimes (but not in this case) a good sign can be found in Bartlett et al. (2020) and Hastie et al. (2020)

[46]:

	param_learning_rate	param_max_depth	mean_test_score	std_test_score
0	0	1	0.581699	0.013072
1	0	4	0.581699	0.013072
2	0	8	0.581699	0.013072
3	0	16	0.581699	0.013072
4	2	1	0.511765	0.125613

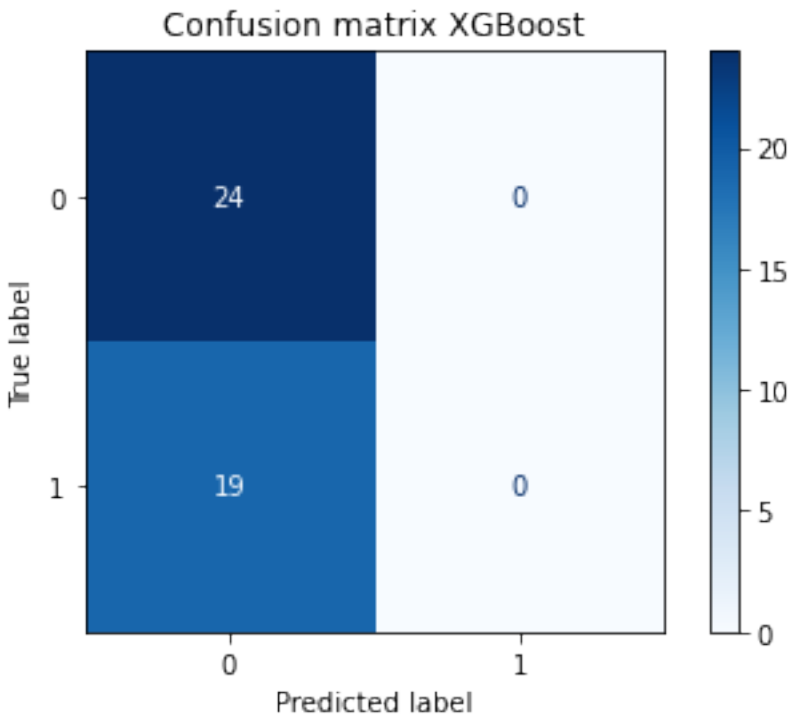
```
Best Parameters: {'learning_rate': 0, 'max_depth': 1}
```

[47]:
```
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=0, max_bin=None,
              max_cat_threshold=None, max_cat_to_onehot=None,
              max_delta_step=None, max_depth=1, 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=1234, ...)
```

The reason for the inclusion of the previous chunk of text is that we intend to show that the chosen model by the grid search is indeed the null model. The accuracy and performance on the test set of XGBoost is as follows, with its corresponding confusion matrix in the figure below, which again shows how the final preferred model is just the null model that always predicts the most common class.

[48]:
```
y_pred = best_xgb.predict(X_test)
accuracy_score(y_pred, y_test)
```

[48]:
```
0.5581395348837209
```



We can see how the gradient boosting always predicts the most common class, and so has a large amount of misclassification errors, and in particular Type II errors.

5 Predictor comparison and Conclusions

The test set accuracies of the three proposed models are the following:

```
Accuracy of RF: 0.5348837209302325
Accuracy of GB: 0.627906976744186
Accuracy of XGB: 0.5581395348837209

F1-loss of RF: 0.3333333333333333
F1-loss of GB: 0.4666666666666667
F1-loss of XGB: 0.0
```

We can see that, in general, they're not very high. This can be due to the large number of NaNs in the data used and them being fitted by interpolation, which may have dampened the signal present in the data. However, a similar analysis removing the data altogether (not present in the final report) showed no improvement. As such, we may conclude that the issue is the large amount of noise in the data where our models are all very good at fitting the training data and therefore extremely vulnerable at overfitting it.

In terms of relative performance, the Gradient Boosting method obtained the best test-set performance, in fact it is considerably better than the other two methods which are essentially the null model (XGBoost), and a noisy version of it (Random Forest). For this reason, the classifier we

would choose would be Gradient Boosting.

Using Gradient Boosting, the best performing model, the ten most relevant recurrence biomarkers are the following peptides:

	Feature	Importance
51	51	0.123774
11	11	0.112551
65	65	0.089577
60	60	0.067424
38	38	0.067323
70	70	0.067106
76	76	0.058383
19	19	0.048530
75	75	0.036543
67	67	0.032089

6 Discussion

Since there is a considerable amount of missing values in the dataset, we resorted to imputing the linear interpolation of the missing entries. This allowed us to work with more information, but it can be argued that some quality was lost (as we saw in the accuracies). More advanced techniques to deal with the missing values for this dataset could be considered, however, the authors claim that an ideal study of this field should be done with a more complete dataset, of with more field knowledge about the biomarkers themselves.

Due to the large amount of noise in the dataset, as seen in the histogram distribution of the pepties in section 2, and the large overfitting tendencies of all three models, other approaches which are more robust to noisy datasets should be considered. For instance, regularisation techniques could help in avoiding strong overftting and the use of **LightGBM** as a more robust ensemble method could be tried out in future studies.

All in all, the best performing model was found to be better than the null model, which is a positive result, despite the fact that its overall performance was far from impressive.

7 References

- Bartlett, P. L. Long, P. M. Lugosi, G. Tsigler, A. (2020) *Benign Overfitting in Linear Regression.* ArXiv.
- Hastie, T. Montanari, A. Rosset, S. Tibshirani, R. J. (2020) *Surprises in high-dimensional Ridgeless Least Squares Interpolation.* ArXiv.

8 Appendix

All the code may be found in this appendix to be reproducible

[49]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import sklearn
from sklearn.metrics import accuracy_score, zero_one_loss, f1_score
from sklearn.metrics import confusion_matrix, classification_report,␣
↪ConfusionMatrixDisplay
from sklearn.model_selection import GridSearchCV
```

[19]:
```
# cancerDat.csv
df1 = pd.read_csv("cancerDat.csv", sep = ";", decimal = ",")
df1 = df1.T
df1 = df1.drop('Unnamed: 0', axis=0)

# cancerInfo.csv
df2 = pd.read_csv("cancerInfo.csv", sep = ";")
df2 = df2.drop(['Unnamed: 0', 'Group'], axis = 1)
df2.index = df2.iloc[:,0]
df2 = df2.drop(['sampleNames'], axis = 1)
```

[7]:
```
print(df1.shape)
print(df2.shape)
display(df1.head())
display(df2.head())
```

```
(129, 102)
(129, 1)
```

	0	1	2	3	4	5	\
NO.REC_1	21.923472	24.442617	19.050562	18.482667	24.086793	20.32946	
NO.REC_2	21.020165	23.649841	18.402413	19.088996	24.710323	21.495392	
NO.REC_3	19.585788	23.736128	18.191527	16.33124	21.917326	20.284533	
NO.REC_4	19.061767	23.374865	17.692775	15.36272	21.484924	18.379603	
NO.REC_5	18.547029	23.039588	19.066973	15.835721	21.339587	19.550809	

	6	7	8	9	...	92	\
NO.REC_1	19.304363	24.270429	18.878984	18.752264	...	19.439382	
NO.REC_2	19.454826	25.807051	19.091796	19.213397	...	20.631064	
NO.REC_3	16.853825	22.661125	18.215654	20.821777	...	19.123832	
NO.REC_4	16.513507	21.401436	18.38696	19.847221	...	17.958307	
NO.REC_5	16.831653	21.776832	17.85408	20.368534	...	18.212854	

	93	94	95	96	97	98	\
NO.REC_1	16.142102	22.858297	22.262118	18.079186	17.151515	20.912124	
NO.REC_2	NaN	22.028998	22.031468	17.101384	18.315637	21.512601	

```
NO.REC_3  16.171227  23.143305  22.334392  17.159968  16.859732  19.076147
NO.REC_4        NaN  19.183961  17.851328  16.564709  15.161135  18.190653
NO.REC_5        NaN  22.228449  21.385404  17.072001  15.071656  18.896095

                99         100        101
NO.REC_1  17.298159  19.097263   21.21211
NO.REC_2  17.100711  18.875548  23.980238
NO.REC_3  15.417028  16.340283  19.810886
NO.REC_4  15.269443  15.411408  18.351433
NO.REC_5  15.147357        NaN   20.28779

[5 rows x 102 columns]
```

```
            sites
sampleNames
NO.REC_1        A
NO.REC_2        B
NO.REC_3        C
NO.REC_4        C
NO.REC_5        C
```

[8]:
```
indices1 = np.array(df1.index).astype(str)
index_array1 = np.zeros(len(indices1))
labels_1 = np.where(np.char.startswith(indices1, 'NO.REC_'), 0, np.where(np.char.
 ↪startswith(indices1, 'REC_'), 1, index_array1))

indices2 = np.array(df2.index).astype(str)
index_array2 = np.zeros(len(indices2))
labels_2 = np.where(np.char.startswith(indices2, 'NO.REC_'), 0, np.where(np.char.
 ↪startswith(indices2, 'REC_'), 1, index_array2))

print("Labels are equal: " + str(np.array_equal(labels_1, labels_2))) # That is␣
 ↪we can define a single response variable y for the two files
y = labels_1
```

```
Labels are equal: True
```

[11]:
```
y = pd.Series(y)
display(y)
```

```
0      0.0
1      0.0
2      0.0
3      0.0
4      0.0
      ...
124    1.0
125    1.0
126    1.0
```

```
127    1.0
128    1.0
Length: 129, dtype: float64
```

[20]:
```
for col in df1:
    df1[col] = pd.to_numeric(df1[col], errors='coerce')
df1.dtypes;
```

[ ]:
```
print(df1.isnull().values.any())
print(sum(df1.isnull()))
df1 = df1.interpolate(method ='linear', limit_direction ='forward')
print(df1.isnull().values.any())
```

[ ]:
```
print(df2.isnull().values.any()) # False -> No nulls
```

[ ]:
```
df1.describe()
```

[ ]:
```
df1.hist(figsize=(12,10))
plt.tight_layout()
plt.show()
```

[ ]:
```
from sklearn.decomposition import PCA
import itertools
pca = PCA(n_components=20)
pca.fit_transform(df1)
#print(pca.explained_variance_ratio_)
#print(list(itertools.accumulate(pca.explained_variance_ratio_)))
plt.plot(list(itertools.accumulate(pca.explained_variance_ratio_)))
```

[ ]:
```
from sklearn.model_selection import train_test_split
np.random.seed(1234)
```

[ ]:
```
from sklearn.ensemble import RandomForestClassifier
rf = RandomForestClassifier(random_state = 1234)
parameters = {'n_estimators' : range(10,130,20),
              'max_features' : (1,2,3,4,5,6,7,8,9,10)
             }
```

[ ]:
```
grid_rf = GridSearchCV(rf, parameters)
grid_rf.fit(X_train, y_train)
```

[ ]:
```
rf_results = pd.DataFrame(grid_rf.cv_results_)
rf_results.filter(regex = '(param.*|mean_t|std_t)') \
    .drop(columns = 'params') \
    .sort_values('mean_test_score', ascending = False) \
    .head(4)
```

[ ]:
```
best_params = grid_rf.best_params_
print("Best Parameters:", best_params)
best_rf = RandomForestClassifier(**best_params, random_state=1234)
best_rf.fit(X_train, y_train)
```

[ ]:
```
y_pred = best_rf.predict(X_test)
accuracy_score(y_pred, y_test)
```

[ ]:
```
confusion_mtx = confusion_matrix(y_test, y_pred)
disp = ConfusionMatrixDisplay(confusion_matrix= confusion_mtx, display_labels=␣
↪None)
disp.plot(cmap= plt.cm.Blues)
plt.title("Confusion matrix Random Forest")
plt.show()
```

[ ]:
```
feature_importances = best_rf.feature_importances_
importance_df = pd.DataFrame({'Feature' : X_train.columns, 'Importance' :␣
↪feature_importances})
print(importance_df.sort_values(by=['Importance'], ascending=False).head(10))
```

[ ]:
```
from sklearn.ensemble import GradientBoostingClassifier
gb = GradientBoostingClassifier(n_estimators=2000, random_state=1234)
parameters = {'max_depth' : (1,4,8,16)}
```

[ ]:
```
grid_gb = GridSearchCV(gb, parameters)
grid_gb.fit(X_train, y_train)
```

[ ]:
```
gb_results = pd.DataFrame(grid_gb.cv_results_)
gb_results.filter(regex = '(param.*|mean_t|std_t)') \
    .drop(columns = 'params') \
    .sort_values('mean_test_score', ascending = False) \
    .head(4)
```

[ ]:
```
y_pred = best_gb.predict(X_test)
accuracy_score(y_pred, y_test)
```

[ ]:
```
best_params = grid_gb.best_params_
print("Best Parameters:", best_params)

train_errors = []
best_gb = GradientBoostingClassifier(**best_params, random_state=1234)
best_gb.fit(X_train, y_train)

for i, y_pred_train in enumerate(best_gb.staged_predict(X_train)):
    train_errors.append(zero_one_loss(y_train, y_pred_train))

plt.plot(np.arange(0,100), train_errors)
```

```
plt.xlabel('Number of Boosting iterations')
plt.ylabel('Misclassification rate')
plt.title('Gradient Boosting Error Evolution')
```

[ ]:
```
y_pred = best_gb.predict(X_test)
accuracy_score(y_pred, y_test)
```

[ ]:
```
confusion_mtx = confusion_matrix(y_test, y_pred)
disp = ConfusionMatrixDisplay(confusion_matrix= confusion_mtx, display_labels=␣
 ↪None)
disp.plot(cmap= plt.cm.Blues)
plt.title("Confusion matrix Gradient Boosting")
plt.show()
```

[ ]:
```
import xgboost
xgb = xgboost.XGBClassifier(n_estimators=2000, random_state=1234)
parameters = {'max_depth' : (1,4,8,16),
              'learning_rate' : [0,2]
              }
```

[ ]:
```
grid_xgb = GridSearchCV(xgb, parameters)
grid_xgb.fit(X_train, y_train)
```

[ ]:
```
xgb_results = pd.DataFrame(grid_xgb.cv_results_)
xgb_results.filter(regex = '(param.*|mean_t|std_t)') \
    .drop(columns = 'params') \
    .sort_values('mean_test_score', ascending = False) \
    .head(5)
```

[ ]:
```
best_params = grid_xgb.best_params_
print("Best Parameters:", best_params)

best_xgb = xgboost.XGBClassifier(**best_params, random_state=1234)
best_xgb.fit(X_train, y_train)
```

[ ]:
```
y_pred = best_xgb.predict(X_test)
accuracy_score(y_pred, y_test)
```

[ ]:
```
confusion_mtx = confusion_matrix(y_test, y_pred)
disp = ConfusionMatrixDisplay(confusion_matrix= confusion_mtx, display_labels=␣
 ↪None)
disp.plot(cmap= plt.cm.Blues)
plt.title("Confusion matrix XGBoost")
plt.show()
```

[ ]:
```
print("Accuracy of RF:", accuracy_score(best_rf.predict(X_test), y_test))
print("Accuracy of GB:", accuracy_score(best_gb.predict(X_test), y_test))
```

```
print("Accuracy of XGB:", accuracy_score(best_xgb.predict(X_test), y_test))
print()
print("F1-loss of RF:", f1_score(best_rf.predict(X_test), y_test))
print("F1-loss of GB:", f1_score(best_gb.predict(X_test), y_test))
print("F1-loss of XGB:", f1_score(best_xgb.predict(X_test), y_test))
```

[ ]:
```
feature_importances = best_gb.feature_importances_
importance_gb = pd.DataFrame({'Feature' : X_train.columns, 'Importance' :␣
↪feature_importances})
print(importance_gb.sort_values(by=['Importance'], ascending=False).head(10))
```

[ ]:
```

```