Project 4 - MT7049 Statistical learning

August Jonasson & Martin Löfström

March 26, 2025

Overview and introduction to data

We aim to build a suitable classification model for binary-labeled breast cancer data[1]. We chose to work with this dataset as it is rather small, has no missing values and its covariates are all of the same type, thus letting us focus primarily on model tuning and selection instead of data wrangling, while also letting us utilise rather computation-heavy algorithms.

The data consists of 569 observations of 30 continuous covariates, each representing some base feature from a digitised image of cell nuclei present within tumorous breast mass. The ten base features of each cell nucleus can be seen from Table 1, and the 30 covariates of our dataset result from the observed **mean (1)**, **standard error (2)** and **worst occurence (3)** of each base feature within an image. All covariates take values in different ranges in $\mathbb{R}^+$. The observations have a binary response $y \in \{-1, 1\}$ corresponding to the cancer being **benign** or **malign** and it is this response that we aim to model. In the information of the dataset it is stated that the covariates are linearly non-separable, and as such, we have to use a statistical method that can handle this suitably. One such family of methods are **tree-based** and we will restrict ourselves to study only these within this report. Specifically, we will fit a gradient-boosted trees model, and use random forests as a performance-comparison.

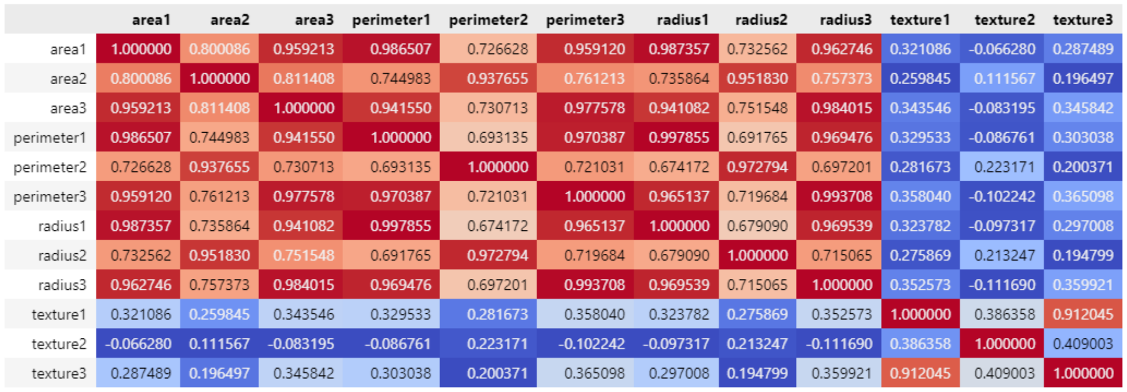
Table 1: Cell nucleus features and descriptions from breast mass image.

Feature	**Description**
Radius	Mean of distances from center to points on the perimeter
Texture	Standard deviation of gray-scale values
Perimeter	Total distance around the cell nucleus
Area	Total area enclosed within the cell nucleus
Smoothness	Local variation in radius lengths
Compactness	(perimeter^2 / area) - 1
Concavity	Severity of concave portions of the contour
Concave points	Number of concave portions of the contour
Symmetry	Symmetry of the cell nucleus
Fractal dimension	Coastline approximation minus 1

[1]found at https://archive.ics.uci.edu/dataset/17/breast+cancer+wisconsin+diagnostic

Exploratory analysis

As previously mentioned, all 30 covariates are derived from a set of 10 base features (three per feature), so, we do expect a considerable colinearity, at least between the covariates stemming from the same base feature (and for covariates stemming from base features which can be expected to be correlated (such as area and perimeter)). Some of the more severe correlations can be seen in Figure 1.



	area1	area2	area3	perimeter1	perimeter2	perimeter3	radius1	radius2	radius3	texture1	texture2	texture3
area1	1.000000	0.800086	0.959213	0.986507	0.726628	0.959120	0.987357	0.732562	0.962746	0.321086	-0.066280	0.287489
area2	0.800086	1.000000	0.811408	0.744983	0.937655	0.761213	0.735864	0.951830	0.757373	0.259845	0.111567	0.196497
area3	0.959213	0.811408	1.000000	0.941550	0.730713	0.977578	0.941082	0.751548	0.984015	0.343546	-0.083195	0.345842
perimeter1	0.986507	0.744983	0.941550	1.000000	0.693135	0.970387	0.997855	0.691765	0.969476	0.329533	-0.086761	0.303038
perimeter2	0.726628	0.937655	0.730713	0.693135	1.000000	0.721031	0.674172	0.972794	0.697201	0.281673	0.223171	0.200371
perimeter3	0.959120	0.761213	0.977578	0.970387	0.721031	1.000000	0.965137	0.719684	0.993708	0.358040	-0.102242	0.365098
radius1	0.987357	0.735864	0.941082	0.997855	0.674172	0.965137	1.000000	0.679090	0.969539	0.323782	-0.097317	0.297008
radius2	0.732562	0.951830	0.751548	0.691765	0.972794	0.719684	0.679090	1.000000	0.715065	0.275869	0.213247	0.194799
radius3	0.962746	0.757373	0.984015	0.969476	0.697201	0.993708	0.969539	0.715065	1.000000	0.352573	-0.111690	0.359921
texture1	0.321086	0.259845	0.343546	0.329533	0.281673	0.358040	0.323782	0.275869	0.352573	1.000000	0.386358	0.912045
texture2	-0.066280	0.111567	-0.083195	-0.086761	0.223171	-0.102242	-0.097317	0.213247	-0.111690	0.386358	1.000000	0.409003
texture3	0.287489	0.196497	0.345842	0.303038	0.200371	0.365098	0.297008	0.194799	0.359921	0.912045	0.409003	1.000000

Figure 1: Some of the most severely correlated covariates from the breast cancer data set.

However, as our focus lies on tree-based methods, where only one covariate at a time is used to split the data, the colinearity will not have an impact on the model fitting. It may, nevertheless, influence the variable importance plot (VIP) (more on this in the analysis section).

Another benefit of using tree-based methods (as opposed to support-vector machines, which have good predictive capabilities and are thus a popular choice for prediction) is that they are insensitive to monotone transformations. Since the selection of the covariate is based on how well it can split a group of observations into two (minimise the split criterion), tree-based methods are invariant to the varying ranges of our covariates and no standardisation is necessary (cf. Hastie, Tibshirani, and Friedman 2009, Table 10.1). In our case this simplifies the analysis considerably as it may be difficult to find a suitable scaling.

Model selection

In the present section, we are to perform model fitting using gradient-boosted trees, and compare with random forests. For both models, we use cross-entropy as our split criterion, which simplifies to the binomial deviance[2]:

$$-\hat{p}\log\hat{p} - (1-\hat{p})\log(1-\hat{p}), \tag{1}$$

where $\hat{p}$ denotes the proportion of one of the classes in a node of the tree (cf. Hastie, Tibshirani, and Friedman 2009, section 9.2.3). This benefits from being monotonously decreasing seen as a function of

[2]since we have binary response.

the margin $yf(x)$ (where $f(x)$ is the function from which sign we predict the response of an observation x); in particular, it will penalise misclassified points (i.e., points with $yf(x) < 0$). This should be contrasted with the square loss generally used for regression, which also benefits from being decreasing for negative margin values, but which will also penalise large values $f(x)$, even if they are correctly classified. Furthermore, the cross-entropy decreases linearly for large negative margin values, making it relatively robust compared to the exponential loss used in AdaBoost, which gives relatively large importance to points with large negative margin value (cf. section 10.6 in Hastie, Tibshirani, and Friedman 2009). Furthermore, for both models we set the amount of covariates to be randomly sampled in each split to $\lfloor\sqrt{30}\rfloor = 5$.

Gradient boosted trees

We begin fitting the gradient-boosted trees model. The progamming was done in Python by implementing the `GradientBoostingClassifier` function from the `scikit-learn` library. [3] We get the following set of hyperparameters to tune:

(i) learning rate, also referred to as shrinkage (ν),

(ii) subsampling fraction (η),

(iii) number of leaves/tree (N), and

(iv) number of trees (M).

In order to determine reasonable values for the parameters we estimate the test error by cross-validation for different number of trees, and different values of the three remaining parameters. Our idea is to find a set of parameter values which yields a small cross-validation error, by optimising the parameters one at a time and saving them for subsequent parameter-optimisations (as opposed to a complete grid-search). Furthermore, we should be able to determine our values in a reasonable amount of time. [4]

To be precise, we first plot the cross-validation error against the number of trees, this time for different values of ν, letting the number of leaves and the subsampling fraction be set to default values (i.e., $N = 4$,[5] $\eta = 1$). Our reason to begin by analysing the learning rate is since the subsampling fraction ought not to influence the test error a lot (Johansson and Ohlsson 2022), and since there is typically not any considerable improvement in test error obtained by changing N from being in the range $4 \leq N \leq 8$ (Hastie, Tibshirani, and Friedman 2009). We then choose a reasonable learning rate ν_0, and again plot the cross-validation error against the number of trees, this time for different values of η, letting as before $N = 4$, and letting $\nu = \nu_0$. We then choose a suitable subsampling fraction η_0, and plot the cross-validation error against the number of trees, this time for different N, letting $\nu = \nu_0$, $\eta = \eta_0$. Finally, we choose the number of leaves $N = N_0$, and find for which M the cross-validation error is minimised, when $\nu = \nu_0$, $\eta = \eta_0$, and $N = N_0$.

[3] https://scikit-learn.org/1.5/modules/generated/sklearn.ensemble.GradientBoostingClassifier.html

[4] e.g., it is known that we can obtain very small test error by picking a very small value ν and a very large value M corresponding to this ν (cf. (Hastie, Tibshirani, and Friedman 2009) for details). However, the large M will require large computational power.

[5] We choose N to be a power of two in order to reduce computational complexity.

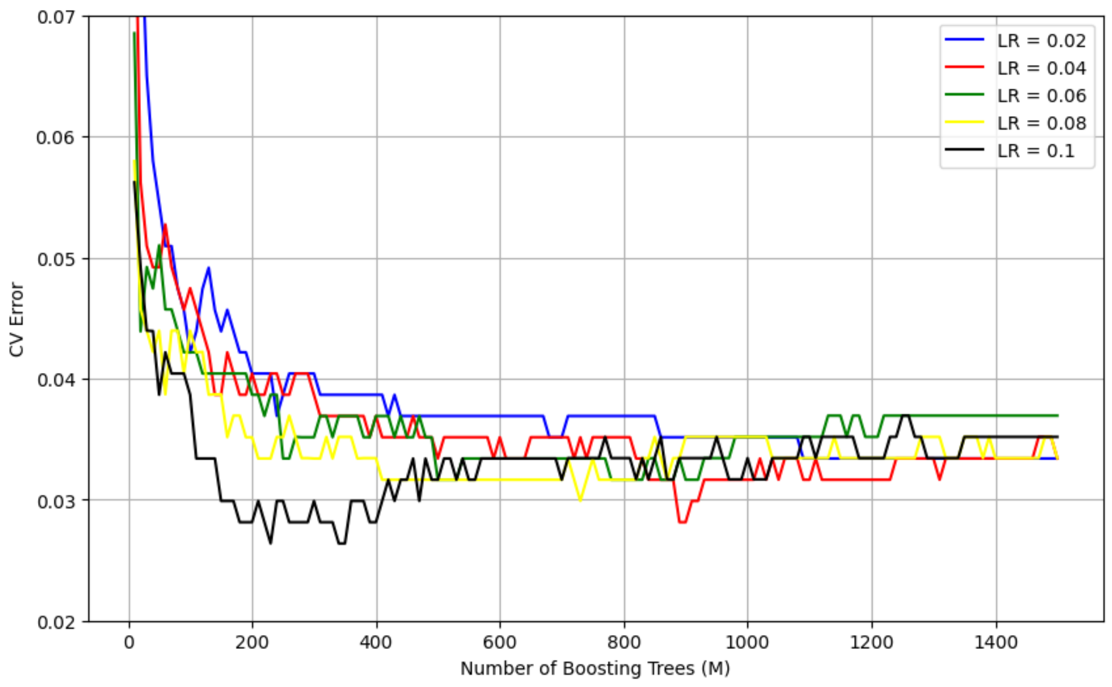


Figure 2: 10-fold cross-validation error against number of gradient-boosted trees for different learning rates LR ($= \nu$). $N = 4$ and $\eta = 1$ are fixed.

In Figure 2, we see that the smallest cross-validation error is obtained for $\nu = 0.1$, for $M \approx 300$. For larger M, the cross-validation error starts to grow, which indicates overfitting. Looking at the models corresponding to the remaining learning rates, only $\nu = 0.02$ show no indication of overfitting. As such, were we to choose between all learning rates except $\nu = 0.02$, we would choose $\nu = 0.1$, as this achieves the lowest error before indications of overfitting. On a separate run of only the model corresponding to $\nu = 0.02$, for even larger M, we saw that the error did not fall below that of the $\nu = 0.1$ model. We thus set $\nu_0 = 0.1$ and move on to determine the subsampling fraction.

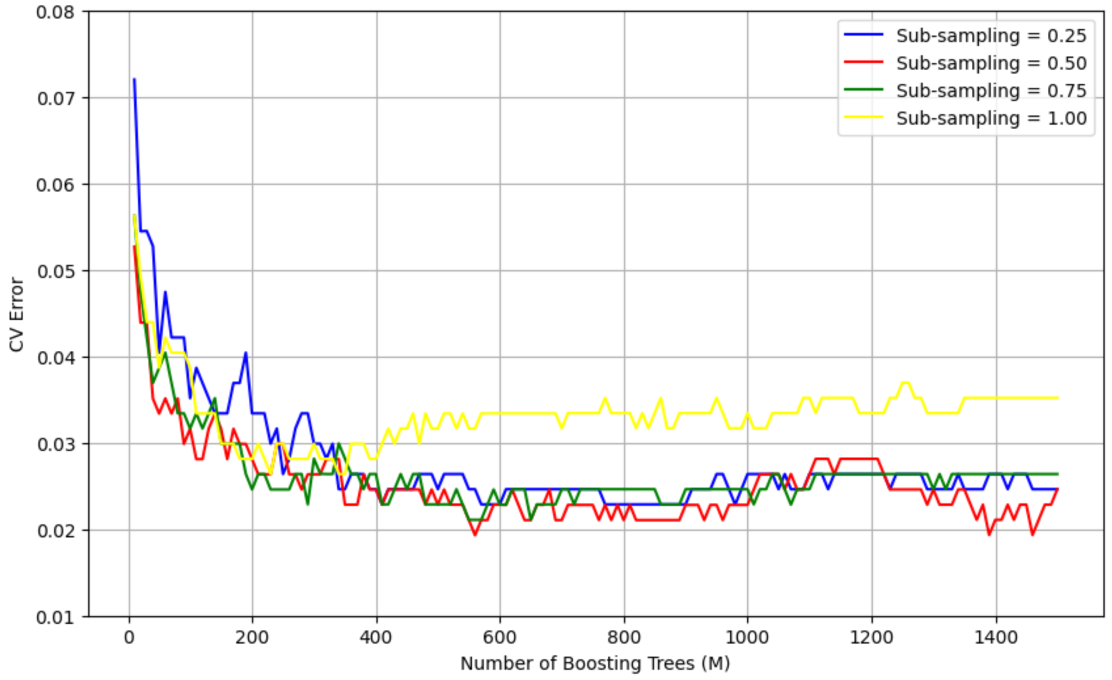


Figure 3: 10-fold cross-validation error against number of gradient-boosted trees for different subsampling fractions η (i.e. stochastic gradient boosting). $\nu_0 = 0.1$, $N = 4$ are fixed.

In Figure 3, we have plotted the cross-validation error for the default subsampling fraction $\eta = 0.5$, and, following a recommendation in Johansson and Ohlsson 2022, we also include the choices $\eta = 0.75$ and $\eta = 1$ in the plot. We have also included $\eta = 0.25$. It is immediately evident that no subsampling at all considerably worsens the prediction. The remaining values on η all perform similarly and as subsampling introduces stochasticity into the optimisation of the model, it is difficult to say that one is clearly better than the other. However, as 0.5 is the usually recommended choice (and indeed, the choice that performs best in our case) we set $\eta_0 = 0.5$.

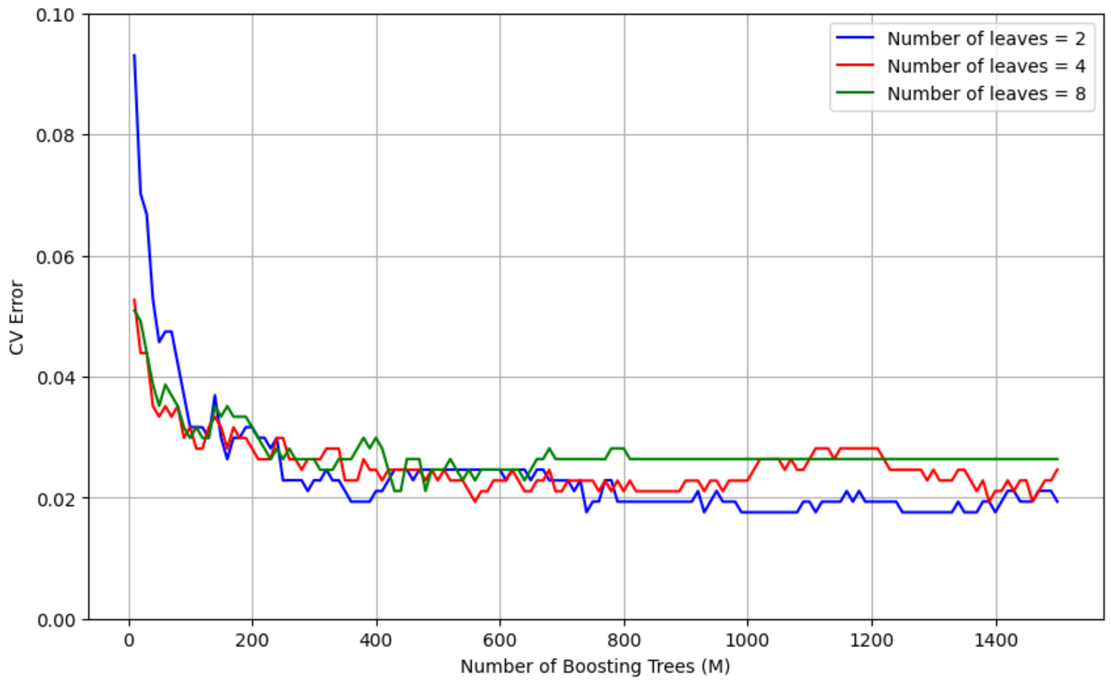


Figure 4: 10-fold cross-validation error against number of gradient-boosted trees for differently sized sub-trees in the form of maximally allowed number of terminal nodes $= N$. $\nu_0 = 0.1$, $\eta_0 = 0.5$ are fixed.

In Figure 4, it is clear that $N = 2$ seems to give the lowest cross-validation error. Although this differs from the usual recommendation of choosing $4 \leq N \leq 8$ (cf. Hastie, Tibshirani, and Friedman 2009), it is not a surprising result, and is in line with results often found in certain applications, such as regression for claim severity (cf. Johansson and Ohlsson 2022).

To conclude, we get:

$$\begin{cases} \nu_0 & = 0.1, \\ \eta_0 & = 0.5, \\ N_0 & = 2. \end{cases}$$

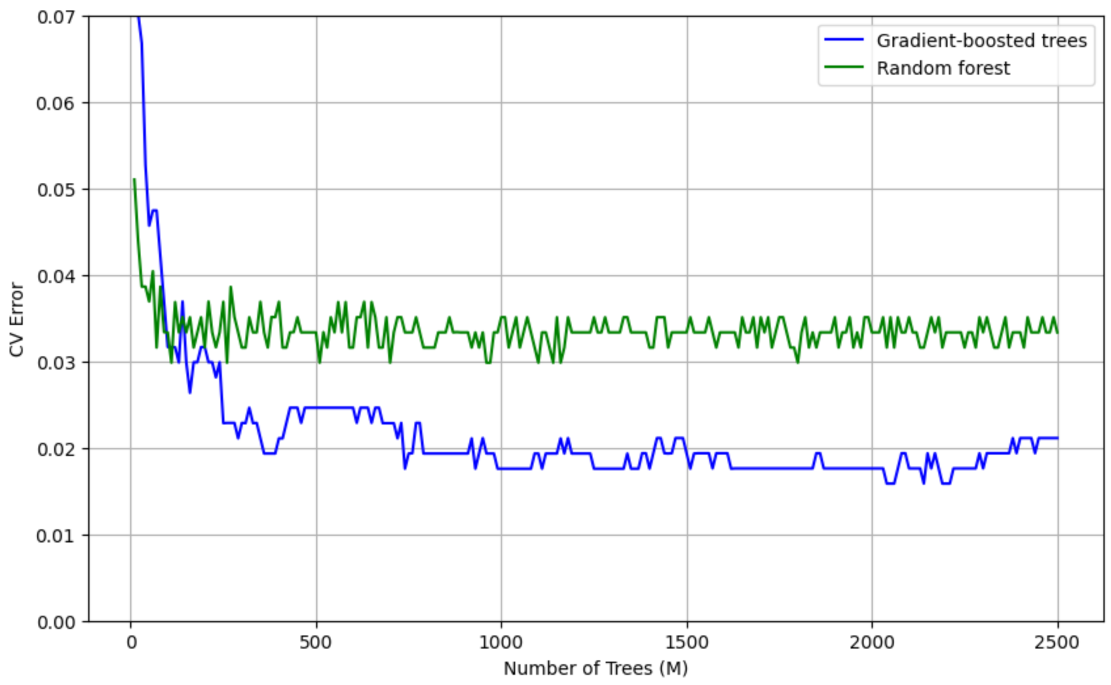


Figure 5: Selected gradient-boosted trees model ($\nu_0 = 0.1$, $\eta_0 = 0.5$, $N_0 = 2$) along with random forest (maximally deep trees allowed, full bootstrap sample size) for 10-fold cross-validation error against number of trees. Minimum achieved at $M_0 = 2040$.

Random forest

We now compare the performance of our gradient-boosted trees model above with another popular tree-based method: random forest, implementing the `RandomForestClassifier` package from the `scikit-learn` library [6] in Python.

One reason to use random forests is that they are easier to apply; we do not need to perform the time-consuming parameter tuning we made in the previous section. Another benefit of random forest is that the trees can be created in parallel as opposed to sequentially (as in gradient-boosting), thus allowing for faster model fitting. Although random forests typically do not perform better than gradient-boosted trees (cf. Hastie, Tibshirani, and Friedman 2009), they may still be preferred if they perform almost as well as gradient-boosted trees. However, we see from Figure 5 that random forests perform considerably worse in our case. It is also worth noting that the performance of random forests stabilises around $M = 100$ (cf. Hastie, Tibshirani, and Friedman 2009), while the gradient-boosted tree model starts overfitting slightly after $M = 2000$.

[6] https://scikit-learn.org/1.5/modules/generated/sklearn.ensemble.RandomForestClassifier.html

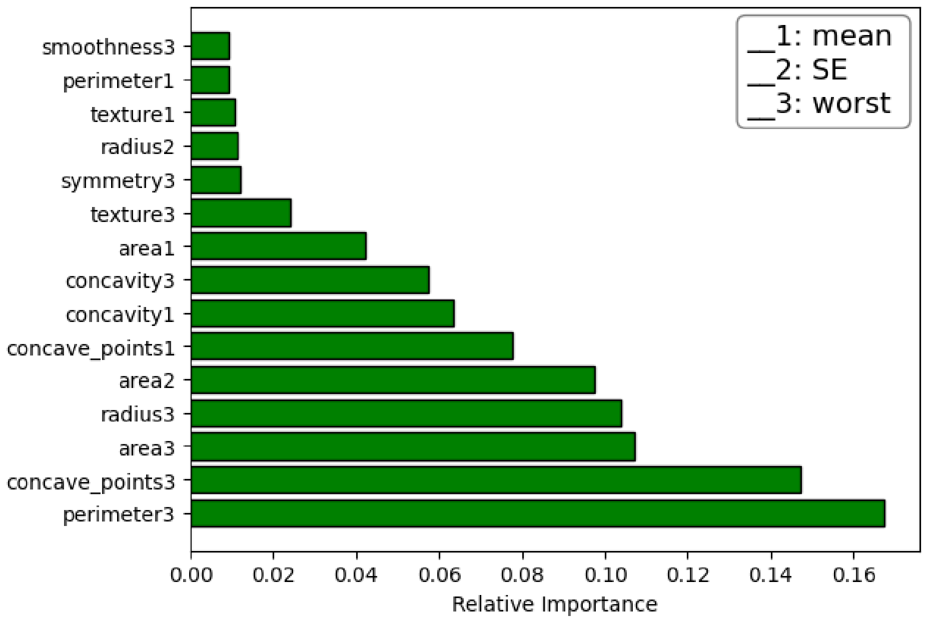


Figure 6: Top 15 (out of 30) cell nuclei covariates in the gradient-boosted model using the relative importance measure (≈ 94 % of total importance).

Analysis and interpretation

Having selected and tuned the gradient-boosted trees model, we now move on to the analysis. In order to interpret the results, we employ the relative importance of the predictor variables (Hastie, Tibshirani, and Friedman 2009, section 10.13.1). In essence, the more often a covariate is used to perform a split in an individual tree, the higher its importance.

We retrieve the relative importance for the covariates using the `feature_importance_` attribute of the `GradientBoostingClassifier` function used previously. These are calculated using the deviance-based importance (equivalent to our split-criterion (1)) within each tree and then averaging this over all trees in the model, [7] and normalising. Figure 6 shows the result. Evidently, roughly half of the total importance in the model is given by the **worst occurence**-covariates: perimeter, number of concave points, area and radius of the cell nuclei. This indicates that extreme values on these covariates especially are more telling than the others when trying to classify the cancer as benign or malign. Eight out of the top 15 most important covariates are of this **worst-occurence (3)** type, five are of the **mean (1)** type, and two are of the **standard error (2)** type.

[7] https://scikit-learn.org/stable/modules/ensemble.html#interpretation-with-feature-importance

When classifying binary-labeled data, we are interested in mapping the tendencies of misclassifications produced by our model: whether our model tends to produce false negatives at a different rate than false positives - false negatives being much more severe in the case of diagnosing cancer. This may be done by splitting the data into training and test data in combination with the use of a two-by-two confusion matrix on the predictions produced from the test data (false negatives/positives being represented outside the main diagonal). However, since the data we are working with consists of only 569 observations, letting 20% (which is the commonly used proportion dedicated to test data) of the observations being test data would give 120 test data points. Knowing the observed cross-validation error of our model to be ≈ 0.17, we can expect to have ≈ 2 misclassified test points. This is clearly not enough to determine a possible preference for the model to produce false negatives over false positives, or vice versa. So, such analysis would be irrelevant.

Relative variable importance may be influenced by the correlation between the variables; a good predictor highly correlated with an even better predictor may appear to have low importance; if a covariate is chosen for a certain split, it is likely that any highly correlated covariate would be almost as good of a choice, but only the better predictor will appear to have an influence. However, this phenomenon is (partly) compensated for by the random sampling of the covariates; for each tree, we sample a subset of the covariates at random and the splits in the tree are based only on these.[8] The grade of compensation would depend on how many covariates that are highly correlated. We will not dig deeper into this issue of the impact of the correlation on the VIP, but note that, although highly correlated, both perimeter$_3$ and area$_3$ are among the three most influential covariates in Figure 6.

Drawbacks and possible improvements

To optimise the parameters in the gradient-boosted tree model properly, it is suitable to perform a grid search on the variables (ν, η, N): to determine a finite set of parameter values (as we did) and plot the misclassification error against M for *each possible combination of parameter values* rather than to optimise the parameters one at a time as we did. Our reason not to make a grid search was time constraints: to make a grid search in a reasonable amount of time would mean we would have to reduce the maximal M considerably, which would make us lose (even more) information on minima of the graphs. Due to this, it is entirely possible, and even likely, that the parameter settings of our choice are not optimal. A possible improvement as such would simply be to use more compute in order to perform a proper grid search.

Moreover, we could have tried to use other methods, such as SVM or neural networks as both of these are known to have good predictive performance. In SVM particularly, we also have the benefit of implicitly taking into account interaction effects between the covariates, something that our tree-based models fail to do on their own.

One of the potentially largest risks with our model is that we do not really know how well it generalises to unseen data. The cross-validation error tends to slightly overestimate the true error (meaning the estimate is conservative), but we still run the risk of overfitting. This would best be investigated by applying the model on new, unseen data. As of now, we cannot really be certain that our model is not overfitted. In order to say something definitive about this we would need new test data.

[8]so that, for the phenomenon to occur, the two highly correlated covariates have to be sampled simultaneously.

Appendix: Python Code

```python
# %%
from ucimlrepo import fetch_ucirepo
import numpy as np
from sklearn.model_selection import train_test_split, cross_val_score
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.ensemble import RandomForestClassifier
from matplotlib import pyplot as plt

# %%

################
# IMPORTING DATA
################

# fetch dataset from uci repo
breast_cancer_wisconsin_diagnostic = fetch_ucirepo(id=17)

# data (as pandas dataframes)
X = breast_cancer_wisconsin_diagnostic.data.features
y = breast_cancer_wisconsin_diagnostic.data.targets

# metadata
print(breast_cancer_wisconsin_diagnostic.metadata)

# variable information
print(breast_cancer_wisconsin_diagnostic.variables)


# %%

#####################################
# CORRELATION PLOT FOR FEATURE SAMPLE
#####################################

# gathering some of the highly correlated features
X_high_corr = X[["area1", "area2", "area3",
                 "perimeter1", "perimeter2", "perimeter3",
                 "radius1", "radius2", "radius3",
                 "texture1", "texture2", "texture3"]]

# correlation plot
corr = X_high_corr.corr()
corr.style.background_gradient(cmap='coolwarm')


# %%

#####################################################
# GRADIENT BOOSTING TREES AND RANDOM FOREST FUNCTIONS
#####################################################

# function for constructing gradient boosted trees
def gradientBoostingClassifier(max_leaf_nodes = 5, max_depth = 10, verbose = 0, n_
    estimators = 100, learning_rate=0.1, subsample=1.0):
    model = GradientBoostingClassifier(
        loss = 'log_loss',
        max_features = 'sqrt',
        max_depth = max_depth,
        max_leaf_nodes =  max_leaf_nodes,
        n_estimators = n_estimators,
        verbose = verbose,
        random_state=0,
        learning_rate=learning_rate,
        subsample=subsample
    )
    return model


# function for constructing the random forests
def randomForestClassifier(n_estimators=100, bootstrap=True, criterion="log_loss"):
    model = RandomForestClassifier(
        n_estimators=n_estimators,
        bootstrap=bootstrap,
        criterion=criterion
    )
    return model

# %%

###########################
# LEARNING RATE / SHRINKAGE
###########################

# specifying the range of number of trees to examine
M_values_lr = range(10, 1510, 10)

# for saving the cv errors of each model
cv_errors_1_lr = []
cv_errors_2_lr = []
cv_errors_3_lr = []
cv_errors_4_lr = []
cv_errors_5_lr = []

# Perform 10-fold CV for each M for different learning rates
for M in M_values_lr:

    model_1 = gradientBoostingClassifier(n_estimators=M, max_leaf_nodes=4, subsample
=1, learning_rate=0.02)
    scores = cross_val_score(model_1, X, y, cv=10, scoring='accuracy')
    mean_error = 1 - np.mean(scores)  # CV error
    cv_errors_1_lr.append(mean_error)

    model_2 = gradientBoostingClassifier(n_estimators=M, max_leaf_nodes=4, subsample
=1, learning_rate=0.04)
    scores = cross_val_score(model_2, X, y, cv=10, scoring='accuracy')
    mean_error = 1 - np.mean(scores)
    cv_errors_2_lr.append(mean_error)

    model_3 = gradientBoostingClassifier(n_estimators=M, max_leaf_nodes=4, subsample
=1, learning_rate=0.06)
    scores = cross_val_score(model_3, X, y, cv=10, scoring='accuracy')
    mean_error = 1 - np.mean(scores)
    cv_errors_3_lr.append(mean_error)

    model_4 = gradientBoostingClassifier(n_estimators=M, max_leaf_nodes=4, subsample
    =1, learning_rate=0.08)
    scores = cross_val_score(model_4, X, y, cv=10, scoring='accuracy')
    mean_error = 1 - np.mean(scores)
    cv_errors_4_lr.append(mean_error)

    model_5 = gradientBoostingClassifier(n_estimators=M, max_leaf_nodes=4, subsample
    =1, learning_rate=0.1)
    scores = cross_val_score(model_5, X, y, cv=10, scoring='accuracy')
    mean_error = 1 - np.mean(scores)
    cv_errors_5_lr.append(mean_error)

# the Figure 2 plot
plt.figure(figsize=(10, 6))
plt.errorbar(M_values_lr, cv_errors_1_lr, label='LR = 0.02', color='blue')
plt.errorbar(M_values_lr, cv_errors_2_lr, label='LR = 0.04', color = 'red')
plt.errorbar(M_values_lr, cv_errors_3_lr, label='LR = 0.06', color = 'green')
plt.errorbar(M_values_lr, cv_errors_4_lr, label='LR = 0.08', color='yellow')
plt.errorbar(M_values_lr, cv_errors_5_lr, label='LR = 0.1', color = 'black')
plt.xlabel('Number of Boosting Trees (M)')
plt.ylabel('CV Error')
plt.ylim(0.02, 0.07)
plt.legend()
plt.grid()
plt.show()

# %%

##############
# SUB-SAMPLING
##############

# specifying the range of number of trees to examine
M_values_ss = range(10, 1510, 10)

# for saving the cv errors of each model
cv_errors_1_ss = []
cv_errors_2_ss = []
cv_errors_3_ss = []
cv_errors_4_ss = []

# Perform 10-fold CV for each M for different sub-sampling sizes
for M in M_values_ss:

    model_1 = gradientBoostingClassifier(n_estimators=M, max_leaf_nodes=4, learning_
    rate=0.1, subsample=0.25)
    scores = cross_val_score(model_1, X, y, cv=10, scoring='accuracy')
    mean_error = 1 - np.mean(scores)
    cv_errors_1_ss.append(mean_error)

    model_2 = gradientBoostingClassifier(n_estimators=M, max_leaf_nodes=4, learning_
    rate=0.1, subsample=0.50)
    scores = cross_val_score(model_2, X, y, cv=10, scoring='accuracy')
    mean_error = 1 - np.mean(scores)
    cv_errors_2_ss.append(mean_error)

    model_3 = gradientBoostingClassifier(n_estimators=M, max_leaf_nodes=4, learning_
    rate=0.1, subsample=0.75)
    scores = cross_val_score(model_3, X, y, cv=10, scoring='accuracy')
    mean_error = 1 - np.mean(scores)
    cv_errors_3_ss.append(mean_error)

    model_4 = gradientBoostingClassifier(n_estimators=M, max_leaf_nodes=4, learning_
    rate=0.1, subsample=1)
    scores = cross_val_score(model_4, X, y, cv=10, scoring='accuracy')
    mean_error = 1 - np.mean(scores)
    cv_errors_4_ss.append(mean_error)


# Figure 3 plot
plt.figure(figsize=(10, 6))
plt.errorbar(M_values_ss, cv_errors_1_ss, label='Sub-sampling = 0.25', color='blue')
plt.errorbar(M_values_ss, cv_errors_2_ss, label='Sub-sampling = 0.50', color = 'red')
plt.errorbar(M_values_ss, cv_errors_3_ss, label='Sub-sampling = 0.75', color = 'green'
    )
plt.errorbar(M_values_ss, cv_errors_4_ss, label='Sub-sampling = 1.00', color='yellow')
plt.xlabel('Number of Boosting Trees (M)')
plt.ylabel('CV Error')
plt.ylim(0.01, 0.08)
plt.legend()
plt.grid()
plt.show()

# %%

##################
# NUMBER OF LEAVES
##################

# specifying the range of number of trees to examine
M_values_nl = range(10, 1510, 10)

# for saving the cv errors of each model
cv_errors_1_nl = []
cv_errors_2_nl = []
cv_errors_3_nl = []


# Perform 10-fold CV for each M for different number of max leaves
for M in M_values_nl:

    model_1 = gradientBoostingClassifier(n_estimators=M, max_leaf_nodes=2, subsample
    =0.50, learning_rate=0.1)
    scores = cross_val_score(model_1, X, y, cv=10, scoring='accuracy')
    mean_error = 1 - np.mean(scores)
    cv_errors_1_nl.append(mean_error)

    model_2 = gradientBoostingClassifier(n_estimators=M, max_leaf_nodes=4, subsample
    =0.50, learning_rate=0.1)
    scores = cross_val_score(model_2, X, y, cv=10, scoring='accuracy')
    mean_error = 1 - np.mean(scores)
    cv_errors_2_nl.append(mean_error)

    model_3 = gradientBoostingClassifier(n_estimators=M, max_leaf_nodes=8, subsample
    =0.50, learning_rate=0.1)
    scores = cross_val_score(model_3, X, y, cv=10, scoring='accuracy')
    mean_error = 1 - np.mean(scores)
    cv_errors_3_nl.append(mean_error)

# Figure 4 plot
plt.figure(figsize=(10, 6))
plt.errorbar(M_values_nl, cv_errors_1_nl, label='Number of leaves = 2', color='blue')
plt.errorbar(M_values_nl, cv_errors_2_nl, label='Number of leaves = 4', color = 'red')
plt.errorbar(M_values_nl, cv_errors_3_nl, label='Number of leaves = 8', color = 'green
    ')
plt.xlabel('Number of Boosting Trees (M)')
plt.ylabel('CV Error')
plt.ylim(0, 0.1)
plt.legend()
plt.grid()
plt.show()

# %%

##################################################
# GBT VS RANDOM FOREST FOR LARGER NUMBER OF TREES
##################################################

# gradient boosted trees models
M_values_gbt = range(10, 2510, 10)
cv_errors_1_gbt = []

for M in M_values_gbt:

    model_gbt = gradientBoostingClassifier(n_estimators=M, max_leaf_nodes=2, subsample
    =0.50, learning_rate=0.1)
    scores = cross_val_score(model_gbt, X, y, cv=10, scoring='accuracy')
    mean_error = 1 - np.mean(scores)
    cv_errors_1_gbt.append(mean_error)

# random forest models
M_values_rf = range(10, 2510 ,10)
cv_errors_rf = []

for M in M_values_rf:

    model_rf = randomForestClassifier(n_estimators=M)
    scores = cross_val_score(model_rf, X, y, cv=10, scoring='accuracy')
    mean_error = 1 - np.mean(scores)
    cv_errors_rf.append(mean_error)

# Figure 5 plot
plt.figure(figsize=(10, 6))
plt.errorbar(M_values_gbt, cv_errors_1_gbt, color='blue', label='Gradient-boosted
    trees')
plt.errorbar(M_values_rf, cv_errors_rf, color='green', label='Random forest')
plt.xlabel('Number of Trees (M)')
plt.ylabel('CV Error')
plt.ylim(0, 0.07)
plt.legend()
plt.grid()
plt.show()

# %%
```

```

####################################################
# FINDING MINIMUM AND CONSTRUCTING OPTIMAL GBT MODEL
####################################################

# retrieving number of trees that yielded min cv error
min_index = np.argmin(cv_errors_1_gbt)

# number_of_trees = M_values_gbt[min_index]
number_of_trees = M_values_gbt[min_index]

# fitting model to chosen params
model_gbt = gradientBoostingClassifier(
    max_leaf_nodes=2,
    n_estimators=number_of_trees,
    learning_rate=0.1,
    subsample=0.5
)

model_gbt.fit(X ,y)

# %%

#########################################
# RETRIEVING RELATIVE FEATURE IMPORTANCES
#########################################

# extracting feature importances
importances = model_gbt.feature_importances_

# sorting features by relative importance
indices = np.argsort(importances)[::-1]

# extra information box for upcoming plot
textstr = "__1: mean \n__2: SE \n__3: worst"
props = dict(boxstyle='round', facecolor='white', alpha=0.5)

# Figure 6 (top 15 most important features in gbt model)
plt.figure()
plt.text(0.134, 12, textstr, fontsize=14, bbox=props)
plt.barh(range(len(importances[:15])), importances[indices][:15], align="center",
    edgecolor='black', color='green')
plt.yticks(range(len(importances[:15])), X.columns[indices][:15])
plt.xlabel("Relative Importance")
plt.show()
```

References

Hastie, T., R. Tibshirani, and J.H. Friedman (2009). *The Elements of Statistical Learning: Data Mining, Inference, and Prediction.* Springer series in statistics. Springer, pp. 309, 346–349, 363, 365, 587, 596. ISBN: 9780387848846. URL: `https://books.google.se/books?id=eBSgoAEACAAJ`.

Johansson, Björn and Esbjörn Ohlsson (2022). "Gradient Boosting Machines and Non-Life Insurance Pricing-Lecture Notes". In: *Available at SSRN 4294965*, pp. 30–32.