

Truls_de_Lange

March 19, 2024

1 Truls de Lange - CA3

1.1 Some circumstance:

I have 4 subjects this semester and am really trying to make a good effort in all of them. A backside to this, is that all subjects are bound to have overlapping assignments with corresponding deadlines right before easter. Thus, I had one midway exam, two major compulsory assignments and two weekly compulsory assignments to get done just the past two weeks - four of which within the first 4 days of this week. If this weren't the case, I would have put much more effort into this assignment to get a score as good as possible. I couldn't do that this time, which I hope you understand. The methods I used here may also not be that thorough or tidy - I'm sorry, but I couldn't really pay that much attention to it. I just had to beat the Kaggle score and get it done.

1.1.1 Imports

```
[4]: # Whenever I needed to import a library at any point in the notebook, I entered ↵it here at the top
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.svm import SVC
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import MinMaxScaler, StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score
from sklearn.feature_selection import SequentialFeatureSelector
from sklearn.ensemble import RandomForestClassifier
```

1.1.2 Reading data

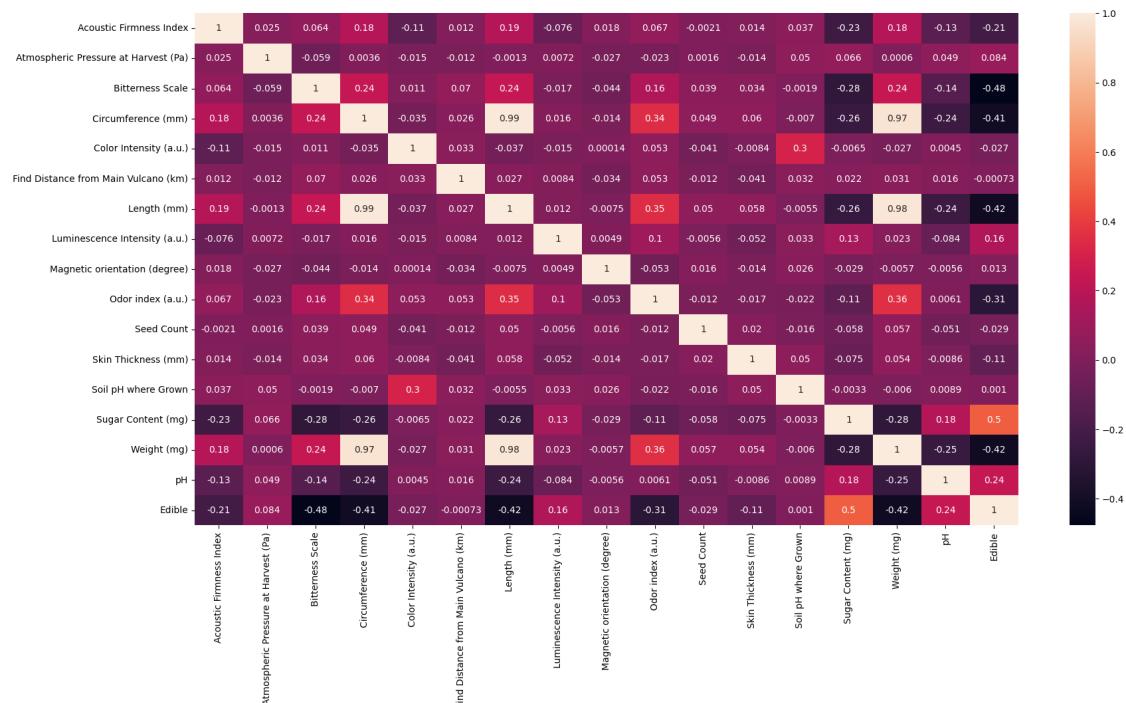
```
[5]: # I read the data in as a pandas dataframe, so that it can be managed with ↵pandas features
data = pd.read_csv("train.csv")
df = pd.DataFrame(data)
df = df.drop(columns='Unnamed: 0')
df.columns
```

```
[5]: Index(['Acoustic Firmness Index', 'Atmospheric Pressure at Harvest (Pa)',  
        'Bitterness Scale', 'Circumference (mm)', 'Color Intensity (a.u.)',  
        'Find Distance from Main Vulcano (km)', 'Length (mm)',  
        'Luminescence Intensity (a.u.)', 'Magnetic orientation (degree)',  
        'Odor index (a.u.)', 'Seed Count', 'Skin Thickness (mm)',  
        'Soil pH where Grown', 'Sugar Content (mg)', 'Weight (mg)', 'pH',  
        'Edible'],  
       dtype='object')
```

1.1.3 Data exploration and visualisation

```
[6]: # Plotting a heatmap of the linear correlation coefficient between all features  
# in the dataset, to check for linear correlation  
plt.figure(figsize=(20, 10))  
sns.heatmap(df.corr(), annot=True)  
plt.plot()  
  
print(len(df))
```

1248



```
[7]: """Since there was little linear correlation between any two features, I plot  
scatter plots of all pairs to check for any other  
kind of correlation"""
```

```

features = list(df)
target = 'Edible'

features.remove('Edible')
features1 = features.copy()
features2 = features.copy()

fig, ax = plt.subplots(20, 6, figsize=(20, 70))

row = 0
col = 0

for feat1 in features1:
    for feat2 in features2:
        if feat2 == feat1:
            continue
        sns.scatterplot(x=feat1, y=feat2, hue=target, data=df, palette={0: \u2022
        \u2022 'coral', 1: 'lightskyblue'}, ax=ax[row][col])
        if col == 5:
            col = 0
            row += 1
            print(f'Row {row} complete')
        else:
            col += 1
    features2.remove(feat1)

plt.tight_layout()
plt.show()

"""
categories = [train_df.]
"""

```

Row 1 complete
Row 2 complete
Row 3 complete
Row 4 complete
Row 5 complete
Row 6 complete
Row 7 complete
Row 8 complete
Row 9 complete
Row 10 complete
Row 11 complete
Row 12 complete
Row 13 complete

Row 14 complete

Row 15 complete

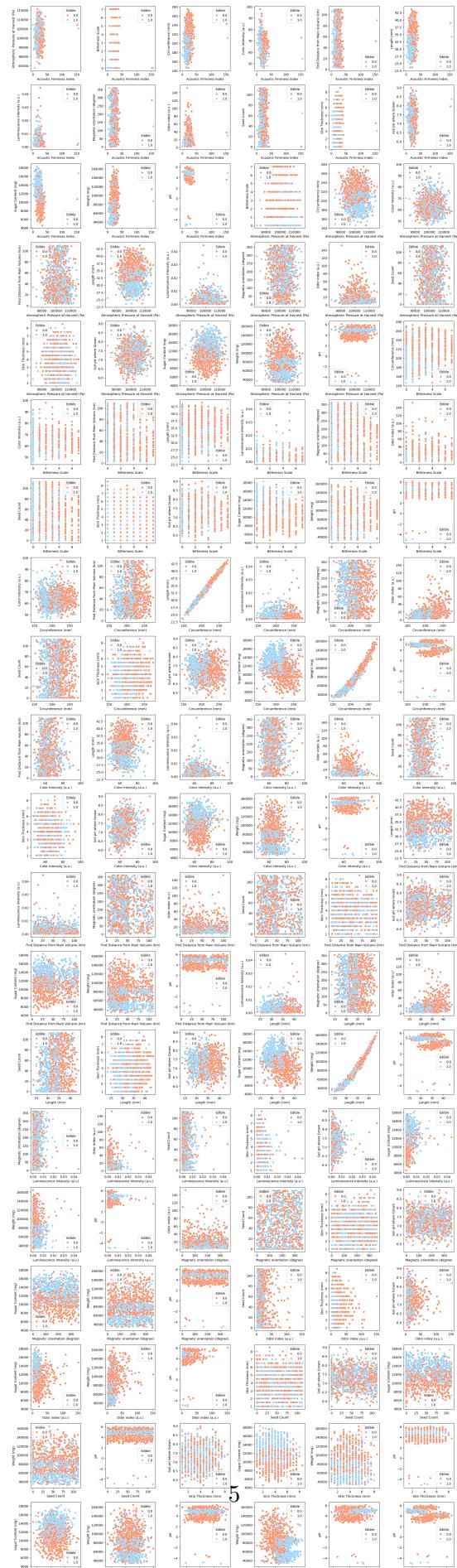
Row 16 complete

Row 17 complete

Row 18 complete

Row 19 complete

Row 20 complete



```
[7]: '\n\ncategories = [train_df.]'\n'
```

Comments on the initial visualisation:

After pairing features and visualising them in scatter plots with target label `Edible` as hue parameter, it seems as if the pH feature, first and foremost, is going to be of great significance to the classification. This is because almost all the plots with pH as one of the features display more or less clearly separable data points. Secondly come the size features (circumference, length, weight...), which also seem to create tidy plots.

One can also see some outliers, above all in the pH-values. This is something I didn't take into account in my calculations, because I thought it would harm the accuracy of the eventual model to get rid of these. This is a decision I grew uncertain of later on, as they may have made both the standardisation, the swapping of NaN-values with mean-values and the final predictions less accurate.

1.1.4 Data cleaning

```
[8]: # X / y split:  
y = pd.DataFrame(df['Edible'])  
X = df.drop(y, axis=1)  
y = y['Edible'].tolist()
```

```
[9]: X[X.isnull().any(axis=1)]
```

```
[9]:   Acoustic Firmness Index Atmospheric Pressure at Harvest (Pa) \
9           18.6             98441.0
25          25.3            102300.0
135         24.4             98459.0
161         21.8             97338.0
190         21.0             109623.0
203         31.4             110062.0
237         16.6             106015.0
280         27.3             101351.0
330          6.2             100398.0
354         18.0             95815.0
411         11.8             103065.0
422          23.0             98213.0
434          NaN             103487.0
440          NaN             100351.0
450          25.0             99958.0
474          20.6             97311.0
557          24.7             103056.0
568          13.6             96623.0
585          7.6              98346.0
761         12.7             97565.0
```

788	17.3	95421.0
810	23.0	102198.0
827	15.8	109257.0
859	NaN	98582.0
896	14.0	98721.0
915	19.9	102155.0
970	31.3	104463.0
986	12.8	103134.0
1009	23.7	101788.0
1033	20.3	90924.0
1063	27.0	101195.0
1105	17.7	103471.0
1124	15.8	102290.0
1132	21.9	101497.0
1146	18.4	98481.0
1162	20.4	106357.0
1212	22.6	102717.0

	Bitterness Scale	Circumference (mm)	Color Intensity (a.u.)	\
9	NaN	193.742185	56.51	
25	4.0	190.527909	60.61	
135	0.0	217.066116	56.70	
161	2.0	196.793798	64.13	
190	0.0	217.672365	58.28	
203	6.0	247.568906	70.78	
237	4.0	223.744843	62.78	
280	2.0	245.449299	62.16	
330	0.0	218.614502	62.09	
354	0.0	210.940636	62.55	
411	4.0	226.188200	81.36	
422	NaN	191.714105	60.34	
434	4.0	188.322992	55.76	
440	0.0	162.971214	60.46	
450	2.0	NaN	66.92	
474	0.0	213.961499	56.04	
557	4.0	248.559479	73.22	
568	1.0	147.912611	72.40	
585	1.0	171.635520	88.89	
761	3.0	215.217767	76.75	
788	3.0	244.434093	64.32	
810	0.0	217.929700	56.76	
827	3.0	222.373678	83.53	
859	0.0	214.689279	55.94	
896	5.0	211.831280	63.53	
915	2.0	218.172347	60.58	
970	1.0	253.669400	71.81	
986	1.0	220.724250	65.83	

1009	1.0	NaN	58.63
1033	0.0	213.938035	69.95
1063	3.0	224.896956	56.85
1105	NaN	170.113774	63.58
1124	NaN	210.795330	60.26
1132	2.0	216.742020	59.52
1146	2.0	217.577700	55.32
1162	1.0	230.559923	64.53
1212	1.0	240.215622	65.03

	Find Distance from Main Vulcano (km)	Length (mm)	\
9	45.007483	30.460301	
25	24.900029	30.716374	
135	24.815873	34.428125	
161	NaN	30.856497	
190	69.797327	34.187210	
203	40.967272	38.856970	
237	26.472882	NaN	
280	79.354862	38.491853	
330	4.736373	34.457676	
354	51.437889	32.705569	
411	83.659900	35.234751	
422	24.955445	30.435190	
434	99.135826	29.436697	
440	34.438603	26.033394	
450	38.414601	36.108872	
474	15.133414	33.573348	
557	82.960324	39.769754	
568	103.505177	23.451799	
585	66.064369	27.057183	
761	51.297853	34.043937	
788	61.459410	39.709118	
810	48.953358	34.487692	
827	46.763876	NaN	
859	44.564549	34.286630	
896	NaN	33.858635	
915	42.191753	34.049559	
970	3.829190	40.290785	
986	17.935211	34.957661	
1009	82.015093	35.250488	
1033	24.477174	NaN	
1063	15.991895	NaN	
1105	46.105552	27.819630	
1124	3.311543	33.701773	
1132	54.215669	34.588742	
1146	82.012560	33.170715	
1162	91.175719	36.107012	

1212 51.436561 38.476410

	Luminescence Intensity (a.u.)	Magnetic orientation (degree)	\
9	0.007231	260.540489	
25	NaN	100.715689	
135	0.000225	160.302385	
161	0.010620	72.248874	
190	0.004681	51.740222	
203	0.002345	333.611746	
237	0.010376	263.399529	
280	0.000640	117.491764	
330	0.004980	81.390160	
354	0.006073	212.767272	
411	0.001196	68.335102	
422	0.000451	319.324289	
434	0.004168	164.072470	
440	0.000720	70.502503	
450	0.006247	285.433610	
474	0.008167	22.250269	
557	0.002554	208.514280	
568	0.004546	169.575803	
585	0.006756	NaN	
761	0.007350	144.872814	
788	0.027688	NaN	
810	0.000163	130.976220	
827	0.004054	97.865194	
859	0.010160	203.174763	
896	0.002089	57.695418	
915	0.000136	306.533553	
970	0.000218	256.752783	
986	0.002530	53.787698	
1009	0.001554	208.565966	
1033	0.002593	331.099150	
1063	0.000237	123.826226	
1105	0.000531	115.569281	
1124	0.001257	100.680008	
1132	0.002604	217.827823	
1146	0.000267	45.654897	
1162	0.001020	NaN	
1212	0.008272	286.196730	

	Odor index (a.u.)	Seed Count	Skin Thickness (mm)	Soil pH where Grown	\
9	6.283100	9.934014	4.5	6.88	
25	10.094517	96.583640	4.0	7.55	
135	NaN	110.052118	5.5	7.07	
161	4.148618	51.922424	4.0	6.78	
190	8.679276	94.442947	NaN	7.43	

203	22.240862	6.767937	7.5	7.74
237	43.789815	6.082008	2.0	7.87
280	11.453696	0.000000	4.0	7.73
330	26.009272	92.676919	3.5	7.04
354	8.246290	29.564740	2.0	NaN
411	63.382994	5.852810	2.5	7.65
422	4.782352	0.000000	2.0	8.08
434	6.831412	10.503935	1.5	6.77
440	5.732340	91.311354	4.5	7.18
450	13.529207	93.910669	5.0	7.68
474	8.781531	61.811010	2.5	7.77
557	11.130196	9.454643	5.0	7.81
568	11.798712	22.511723	3.5	NaN
585	15.040867	106.452684	5.0	7.75
761	16.544032	85.969285	2.0	7.93
788	58.732013	64.148965	3.0	7.18
810	10.306349	84.718283	1.5	6.69
827	33.092417	36.835411	6.5	8.63
859	2.739507	15.321905	5.5	6.96
896	15.061711	78.268009	3.0	7.80
915	6.631049	81.932015	1.0	7.27
970	19.309590	NaN	6.0	7.66
986	15.824135	0.000000	5.5	6.86
1009	24.820283	0.000000	1.5	7.61
1033	6.815152	24.157196	5.0	7.80
1063	12.424632	14.998318	3.0	6.02
1105	4.434950	27.037439	3.0	7.28
1124	6.083661	83.167741	4.0	7.18
1132	NaN	66.932591	4.0	6.81
1146	18.547906	96.998290	5.5	6.71
1162	6.653647	35.543018	6.0	8.16
1212	24.850870	108.791301	2.5	NaN

	Sugar Content (mg)	Weight (mg)	pH
9	11982.0	49288.1826	4.880000
25	12719.0	54448.7826	3.400888
135	9821.0	79927.1826	4.940000
161	16100.0	58319.1826	4.790000
190	16826.0	86009.1826	5.040000
203	8096.0	NaN	3.849393
237	14125.0	84148.7826	3.392290
280	8754.0	NaN	4.090321
330	NaN	84379.1826	4.970000
354	11727.0	71459.1826	4.960000
411	13239.0	88069.1826	NaN
422	13907.0	54842.1826	4.860000
434	13439.0	49526.1826	4.970000

440	13151.0	40103.1826	4.970000
450	10178.0	95596.3026	3.588140
474	11146.0	75254.1826	NaN
557	10206.0	124487.0226	NaN
568	12718.0	34697.1826	5.230000
585	12779.0	35430.1826	5.290000
761	14550.0	69563.1826	NaN
788	11907.0	123846.2226	4.339953
810	NaN	82257.1826	4.960000
827	11010.0	89549.7426	4.583286
859	13255.0	75659.1826	4.890000
896	14377.0	74937.1826	5.080000
915	11736.0	NaN	5.130000
970	8524.0	130367.0226	4.103869
986	12180.0	NaN	4.031517
1009	12617.0	81403.1826	5.770722
1033	15056.0	77434.1826	4.880000
1063	10505.0	84472.7826	3.633814
1105	12480.0	42718.1826	4.910000
1124	13055.0	76409.1826	4.890000
1132	13420.0	81501.1826	5.060000
1146	NaN	72933.5826	4.267374
1162	9658.0	91667.9826	3.991340
1212	6439.0	106196.1426	4.263931

1.1.5 Data preprocessing and visualisation

[14]: *"""As shown above, there are very many rows with NaN-values scattered over many different columns. I thus see removing rows and/or columns with NaN-values as inconvenient, considering the fairly small size of the dataset. I chose the mean-strategy instead, which I thought would be a good substitute - although I see now, that I maybe should have taken care of the outliers first then, to get a cleaner mean.*

I know I could have added some visualisation of the importance of the selected features, but I didn't have time. I also ended up discarding the feature selection, because the model didn't benefit from it."""

```
# Feature imputing:
imputer = SimpleImputer(missing_values=np.nan, strategy='mean') # Don't want to remove any rows, because the train set is very small
imputer.fit(X)
X = imputer.transform(X)
X = pd.DataFrame(X, columns=['Acoustic Firmness Index', 'Atmospheric Pressure at Harvest (Pa)', 'Bitterness Scale', 'Circumference (mm)', 'Color Intensity (a.u.)',
```

```

'Find Distance from Main Volcano (km)', 'Length (mm)',
'Luminescence Intensity (a.u.)', 'Magnetic orientation (degree)',
'Odor index (a.u.)', 'Seed Count', 'Skin Thickness (mm)',
'Soil pH where Grown', 'Sugar Content (mg)', 'Weight (mg)', 'pH'])

# Mean and std computation:
mean = X.mean()
std = X.std()

# Train / test split:
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25,
    ↪random_state=42) # I chose a .25 share test set, because I thought it seemed
    ↪like a good share
X_eval = pd.DataFrame(pd.read_csv('test.csv').drop('Unnamed: 0', axis=1))

```

Random Forest:

As many of the features seemed to show some kinds of non-linear correlation, I thought of the random forest and SVM classifiers, which can make non-linear decision boundaries. First out, I tried various hyperparameters for the random forest:

```

[ ]: """I decided to try random forest, because it's effective and deals well with
    ↪non-linear decision boundaries. As seen in the initial
    visualisation, the two classes are namely separated non-linearly. I also
    ↪experimented with kNN before this, and Kernel SVM afterwards, but
    since it was strictly threatened with rejection for submissions that didn't
    ↪contain only the MOST NECESSARY, I deleted the lot of it. Still think it was
    an important part of the process though."""
import csv

with open('random_forest.csv', 'w', newline='') as f:
    csv.writer(f).writerow(['Estimators', 'Jobs', 'Depth', 'Criterion',
    ↪'Accuracy Train', 'Accuracy Test'])

# I created a loop that takes ages, but computes a thorough list over
    ↪parameters and performance on the test set and stores it in a .csv-file
    for estimators in range(700, 1401, 100):
        print(f'{estimators} estimators:')
        for jobs in range(3, 7):
            print(f'{jobs} jobs:')
            for depth in range(2, 12, 2):
                print('Depth: ', depth)
                rf = RandomForestClassifier(n_estimators=estimators,
    ↪random_state=42, n_jobs=jobs, max_depth=depth, criterion='gini')
                rf.fit(X_train, y_train)

            y_pred_t = rf.predict(X_train)

```

```

accuracy_train = accuracy_score(y_pred_t, y_train)

y_pred = rf.predict(X_test)
accuracy_test = accuracy_score(y_pred, y_test)

csv.writer(f).writerow([estimators, jobs, depth, crit, ↴
accuracy_train, accuracy_test])

```

After looking through the resulting .csv-file, i found that 1200-1400 estimators, 3 jobs and a max depth of 10 were the best parameters.

```
[26]: # Testing feature selection on the best performing RandomForest:
knn = KNeighborsClassifier(n_neighbors=5, n_jobs=10)

sfs = SequentialFeatureSelector(knn, n_features_to_select=5, ↴
direction="forward", n_jobs=4)
sfs = sfs.fit(X_train, y_train)

X_train_sfs = sfs.transform(X_train)
X_test_sfs = sfs.transform(X_test)
X_eval_sfs = sfs.transform(X_eval)
for estimator in range(1000, 10001, 1000):
    for jobs in range(3, 13, 3):
        for depth in range(4, 17, 4):
            rf2 = RandomForestClassifier(n_estimators=estimator, ↴
random_state=42, n_jobs=jobs, max_depth=depth, criterion='gini')
            rf2.fit(X_train_sfs, y_train)
            y_pred = rf2.predict(X_test_sfs)
            accuracy = accuracy_score(y_pred, y_test)
            print(f'Estimators: {estimator}, jobs: {jobs}, Max depth: {depth}, ↴
accuracy: {accuracy}')

# It didn't work out well
```

Estimators: 1000, jobs: 3, Max depth: 4, accuracy: 0.8621794871794872
Estimators: 1000, jobs: 3, Max depth: 8, accuracy: 0.8878205128205128
Estimators: 1000, jobs: 3, Max depth: 12, accuracy: 0.8878205128205128
Estimators: 1000, jobs: 3, Max depth: 16, accuracy: 0.8846153846153846
Estimators: 1000, jobs: 6, Max depth: 4, accuracy: 0.8621794871794872
Estimators: 1000, jobs: 6, Max depth: 8, accuracy: 0.8878205128205128

KeyboardInterrupt Cell In[26], line 14 <pre> 12 for depth in range(4, 17, 4): 13 rf2 = RandomForestClassifier(n_estimators=estimator, ↴ random_state=42, n_jobs=jobs, max_depth=depth, criterion='gini') ---> 14 rf2.fit(X_train_sfs, y_train) </pre>	Traceback (most recent call last)
---	-----------------------------------

```

15     y_pred = rf2.predict(X_test_sfs)
16     accuracy = accuracy_score(y_pred, y_test)

File ~\AppData\Local\miniconda3\envs\dat200_env\lib\site-packages\sklearn\base.py:1351, in _fit_context.<locals>.decorator.<locals>.wrapper(estimator, *args, **kwargs)
1344     estimator._validate_params()
1346 with config_context(
1347     skip_parameter_validation=
1348         prefer_skip_nested_validation or global_skip_validation
1349     )
1350 ):
-> 1351     return fit_method(estimator, *args, **kwargs)

File ~\AppData\Local\miniconda3\envs\dat200_env\lib\site-packages\sklearn\ensemble\_forest.py:489, in BaseForest.fit(self, X, y, sample_weight)
478 trees = [
479     self._make_estimator(append=False, random_state=random_state)
480     for i in range(n_more_estimators)
481 ]
483 # Parallel loop: we prefer the threading backend as the Cython code
484 # for fitting the trees is internally releasing the Python GIL
485 # making threading more efficient than multiprocessing in
486 # that case. However, for joblib 0.12+ we respect any
487 # parallel_backend contexts set at a higher level,
488 # since correctness does not rely on using threads.
--> 489 trees = Parallel(
490     n_jobs=self.n_jobs,
491     verbose=self.verbose,
492     prefer="threads",
493 )(
494     delayed(_parallel_build_trees)(
495         t,
496         self.bootstrap,
497         X,
498         y,
499         sample_weight,
500         i,
501         len(trees),
502         verbose=self.verbose,
503         class_weight=self.class_weight,
504         n_samples_bootstrap=n_samples_bootstrap,
505         missing_values_in_feature_mask=missing_values_in_feature_mask,
506     )
507     for i, t in enumerate(trees)
508 )
510 # Collect newly grown trees

```

```

511 self.estimators_.extend(trees)

File \[+\]
  ↵~\AppData\Local\miniconda3\envs\dat200_env\lib\site-packages\sklearn\utils\parallel.py:67, in Parallel.__call__(self, iterable)
    62 config = get_config()
    63 iterable_with_config = (
    64     (_with_config(delayed_func, config), args, kwargs)
    65     for delayed_func, args, kwargs in iterable
    66 )
---> 67 return super().__call__(iterable_with_config)

File \[+\]
  ↵~\AppData\Local\miniconda3\envs\dat200_env\lib\site-packages\joblib\parallel.py:1952, in Parallel.__call__(self, iterable)
1946 # The first item from the output is blank, but it makes the interpreter
1947 # progress until it enters the Try/Except block of the generator and
1948 # reach the first `yield` statement. This starts the asynchronous
1949 # dispatch of the tasks to the workers.
1950 next(output)
-> 1952 return output if self.return_generator else list(output)

File \[+\]
  ↵~\AppData\Local\miniconda3\envs\dat200_env\lib\site-packages\joblib\parallel.py:1595, in Parallel._get_outputs(self, iterator, pre_dispatch)
1592     yield
1594     with self._backend.retrieval_context():
-> 1595         yield from self._retrieve()
1597 except GeneratorExit:
1598     # The generator has been garbage collected before being fully
1599     # consumed. This aborts the remaining tasks if possible and warn
1600     # the user if necessary.
1601     self._exception = True

File \[+\]
  ↵~\AppData\Local\miniconda3\envs\dat200_env\lib\site-packages\joblib\parallel.py:1707, in Parallel._retrieve(self)
1702 # If the next job is not ready for retrieval yet, we just wait for
1703 # async callbacks to progress.
1704 if ((len(self._jobs) == 0) or
1705     (self._jobs[0].get_status(
1706         timeout=self.timeout) == TASK_PENDING)):
-> 1707     time.sleep(0.01)
1708     continue
1710 # We need to be careful: the job list can be filling up as
1711 # we empty it and Python list are not thread-safe by
1712 # default hence the use of the lock

```

`KeyboardInterrupt`:

1.1.6 Kaggle submission

```
[29]: """The .csv-list over random forest performances indicated that 1200-1400 estimators, 3 jobs and max depth 10 would do. I then made this code to create .csv-lists of predictions for different random forests with nearby hyperparameters. Surprisingly, the first one I uploaded - 1200 estimators, 3 jobs, 10 max depth - still holds the record amongst my submissions with .91737, although I did upload several models with the parameters tweaked a bit to check for wiggle room."""

rf_final = RandomForestClassifier(n_estimators=1200, random_state=42, n_jobs=3, max_depth=10)

rf_final.fit(X, y)
y_pred = rf_final.predict(X_eval)

rf_df = pd.DataFrame(y_pred, columns=['Edible'])
rf_df.to_csv(f'RandomForest_1200_3_10_sfs.csv', index_label='index')
```