Problem introduction

We explore a scikit-learn dataset concerning malignant and begnin breast cancers. Our dataset consists of a feature vector of 30 features and 1 class which can be 0 (Malignant) or 1 (Benign). These features have been extracted from digitized images of cells of breast mass. Thus they describe characteristics of the cell nuclei.

Some of the features

- radius (mean of distances from center to points on the perimeter)
- perimeter
- area
- concavity (severity of concave portions of the contour)
- smoothness (local variation in radius lengths)

Target classes

- Malignant (0)
- Benign (1)

Our goal is to train a machine learning algorithm to classify -given the 30 features- unseen samples as either malignant or benign.

The initial choosen Machine learning classifier is Support Vector Machines. We then explore Random Forests and Logistic Regression to see if we can get to a better accuracy.

The model build here performs with a **98.25%** accuracy after tweaking some of the hyperparameters beating in the process the result of a recent paper (97.31%). This shows how efficient and state-of-the-art scikit-learn really is. Moreover, after hyperparameter tuning it was found that linear kernel performed better than the paper's RBF kernel,

Machine Learning Algorithm Pipeline

1. Question and the required data
2. Acquire the data
3. Data Analysis
4. Prepare the data for the ML model
5. Train the model
6. Test the model
7. Evaluate the model
8. Interpret the model and report results visually and numerically
9. Adjust the model

In [1]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import collections
import random
sns.set()

# Get the data
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split, GridSearchCV, RandomizedSearchCV
from sklearn.svm import SVC
from sklearn.metrics import confusion_matrix,accuracy_score,f1_score,precision_score,recall_score
from sklearn.preprocessing import MinMaxScaler, StandardScaler
from sklearn.pipeline import Pipeline
from sklearn.decomposition import PCA
```

Data Analysis

Typically we have these five stages

1. Tidy Data
2. Missing Values
3. Outliers
4. Class-Imbalance Problem
5. Feature Correlation: visualized using pairplots (optionally one can use a heatmap for visualization)

Tidy data

In [2]:
```
features = load_breast_cancer()

# Lets see what the dataset offers us
print(features.keys(), '\n')

# Temporarily save the labels
classes = features['target']

# Transform them to pandas dataframe
features = pd.DataFrame(data=features['data'], columns=features['feature_names'])

# Add the labels
features['class'] = classes

print("Our dataset has {} samples and {} features(including the label)".format(*features.shape))

features.sample(3)
```

```
dict_keys(['data', 'target', 'target_names', 'DESCR', 'feature_names', 'filename'])

Our dataset has 569 samples and 31 features(including the label)
```

Out[2]:

	mean radius	mean texture	mean perimeter	mean area	mean smoothness	mean compactness	mean concavity	mean concave points	mean symmetry	mean fractal dimension	...	worst texture	worst perimeter	wor ar
278	13.59	17.84	86.24	572.3	0.07948	0.04052	0.01997	0.01238	0.1573	0.05520	...	26.10	98.91	739
405	10.94	18.59	70.39	370.0	0.10040	0.07460	0.04944	0.02932	0.1486	0.06615	...	25.58	82.76	472
102	12.18	20.52	77.22	458.7	0.08013	0.04038	0.02383	0.01770	0.1739	0.05677	...	32.84	84.58	547

3 rows × 31 columns

Now that we have a nice pandas dataframe we can continue to the next steps.

Missing Values & Outliers

A great way to tell if we have missing values or outliers is to compute a summary statistics.

In [3]:
```
features.describe()
```

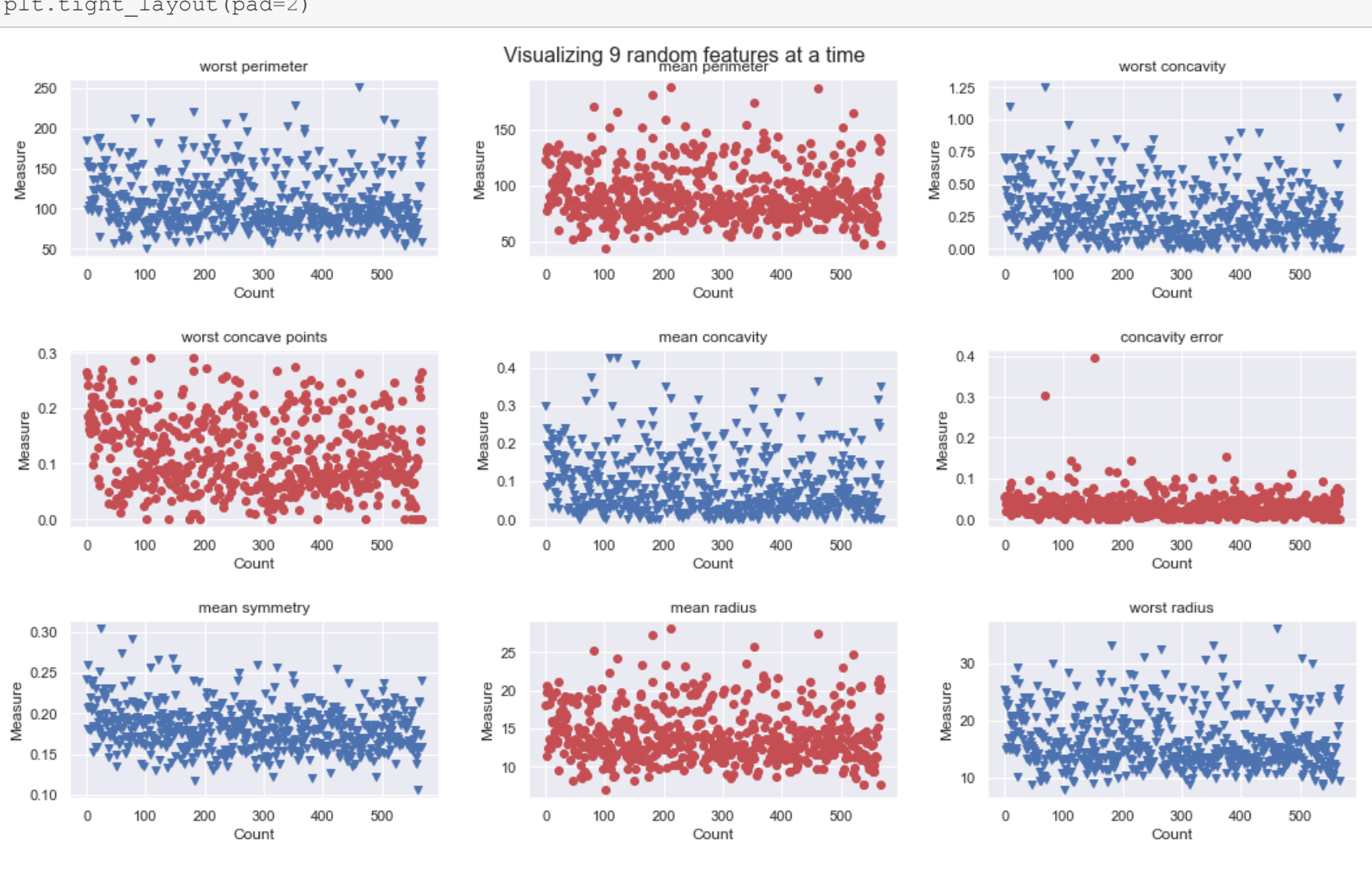
Out[3]:

	mean radius	mean texture	mean perimeter	mean area	mean smoothness	mean compactness	mean concavity	mean concave points	mean symmetry	mean fractal dimension	...
count	569.000000	569.000000	569.000000	569.000000	569.000000	569.000000	569.000000	569.000000	569.000000	569.000000	...
mean	14.127292	19.289649	91.969033	654.889104	0.096360	0.104341	0.088799	0.048919	0.181162	0.062798	...
std	3.524049	4.301036	24.298981	351.914129	0.014064	0.052813	0.079720	0.038803	0.027414	0.007060	...
min	6.981000	9.710000	43.790000	143.500000	0.052630	0.019380	0.000000	0.000000	0.106000	0.049960	...
25%	11.700000	16.170000	75.170000	420.300000	0.086370	0.064920	0.029560	0.020310	0.161900	0.057700	...
50%	13.370000	18.840000	86.240000	551.100000	0.095870	0.092630	0.061540	0.033500	0.179200	0.061540	...
75%	15.780000	21.800000	104.100000	782.700000	0.105300	0.130400	0.130700	0.074000	0.195700	0.066120	...
max	28.110000	39.280000	188.500000	2501.000000	0.163400	0.345400	0.426800	0.201200	0.304000	0.097440	...

8 rows × 31 columns

We can furthermore visualize some of the features. This will also help in getting comfortable with data which is an important step in working with data.

In [4]:
```
# Randomly pick 9 features to visualize
plot_features = random.sample(list(features.columns.drop('class')), 9)
plot_features = features[plot_features]

plt.figure(figsize=(15,9))
plt.suptitle("Visualizing 9 random features at a time", fontsize=16)

for i in range(1, plot_features.shape[1]+1):
    # Add a subplot: 3 rows, 3 cols and i represents the index
    plt.subplot(3,3,i)

    # Get the current plot
    figure = plt.gca()

    # Set the title of the currentplot
    figure.set_title(plot_features.columns[i-1])

    # Add some variety to plots
    if i%2==0:
        plt.plot(plot_features.iloc[:, i-1], 'ro')
    else:
        plt.plot(plot_features.iloc[:, i-1], 'v')
    plt.xlabel("Count"); plt.ylabel("Measure")

# Give some padding to aboid overlap in graphs
plt.tight_layout(pad=2)
```



Generally speaking, we have no outliers. Our dataset also has no missing values.

Class-Imbalance Problem

Shedding some light on the problem here

In [5]:
```
print("Class imbalance problem:")
print(features['class'].value_counts()/features.shape[0])
```

```
Class imbalance problem:
1    0.627417
0    0.372583
Name: class, dtype: float64
```

The classes are more or less balanced.

Feature Correlation

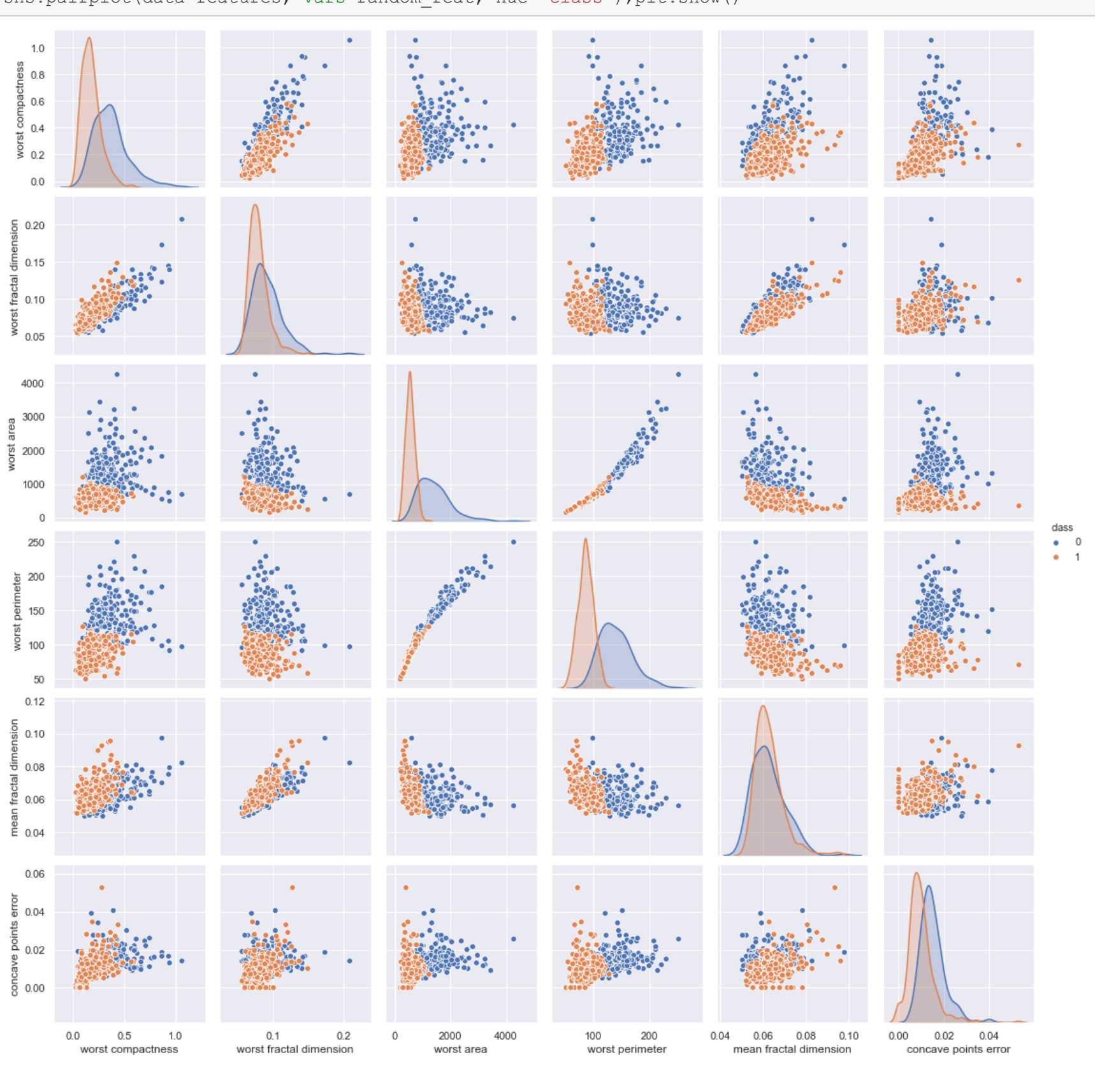
We can nicely represent this by using a pairplot (alternatively we could also build a heatmap).

Variables tweaked:

- hue: maps plot aspects to different colors. In this case we map two different colors to the two possible classes
- vars: defines the list of features to use, otherwise it uses all the numeric datatypes

The diagonal (which plots each variable against itself is actually showing the distribution of each variable.

In [6]:
```
# Pick 6 random features to plot this
random_feat = random.sample(list(features.columns.drop('class')), 6)
sns.pairplot(data=features, vars=random_feat, hue='class');plt.show()
```



We can definitely tell that some features are correlated. But most don't seem to be correlated.

We also see that the values for malignant cases values are actually higher (the radius for instance). Also, our data seems to be well-separated.

It is very important to get comfortable with the data early on. This can be realized with such great plots.

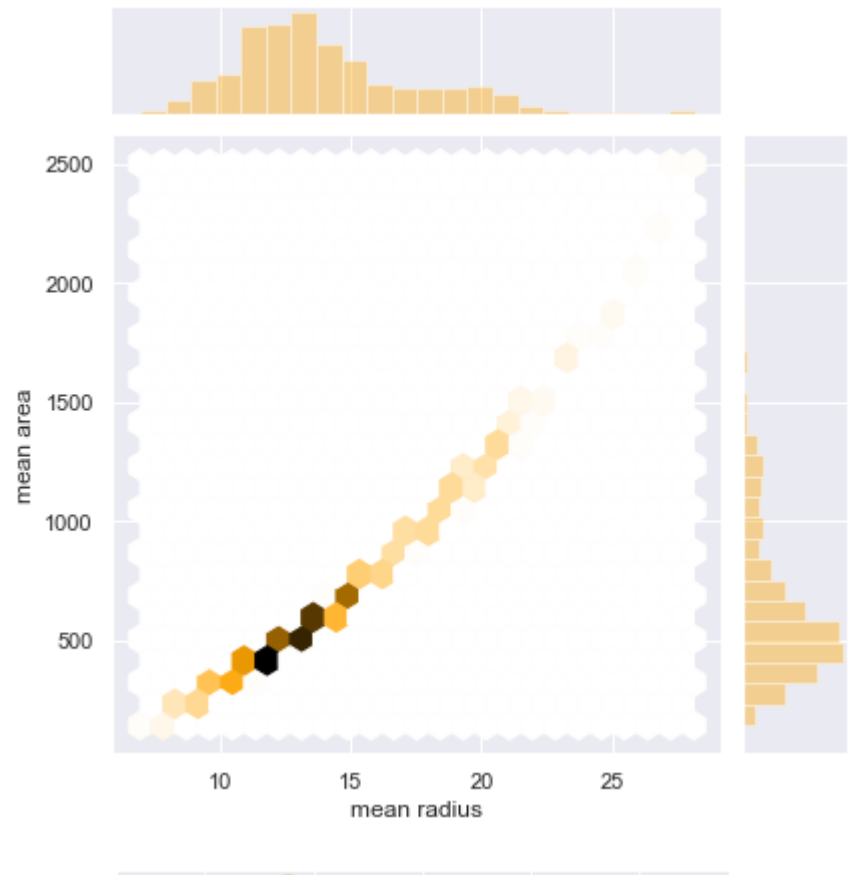
More plots on individual features

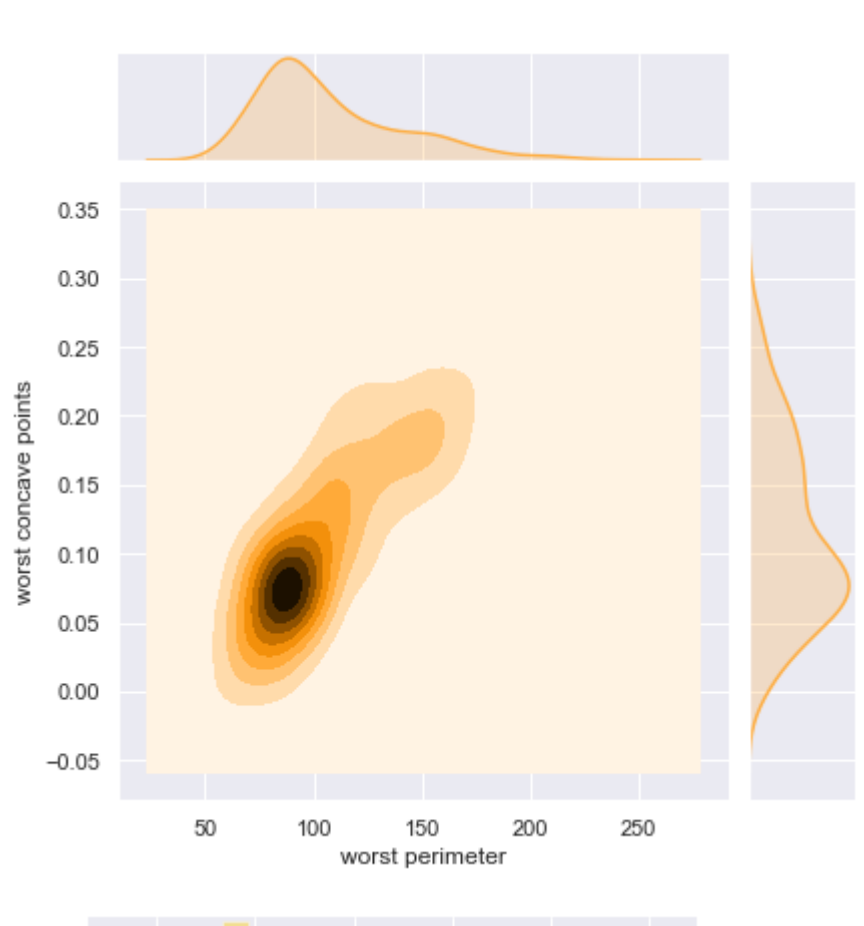
Do some more great plots and check more specifically the correlation individual features.

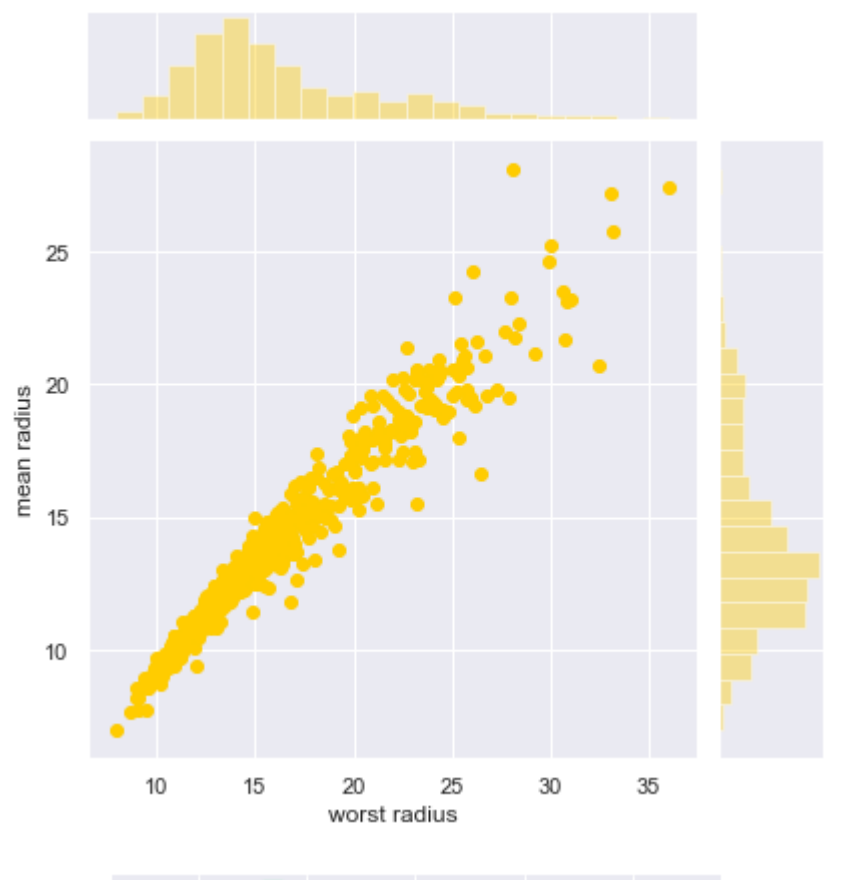
Joint Plot

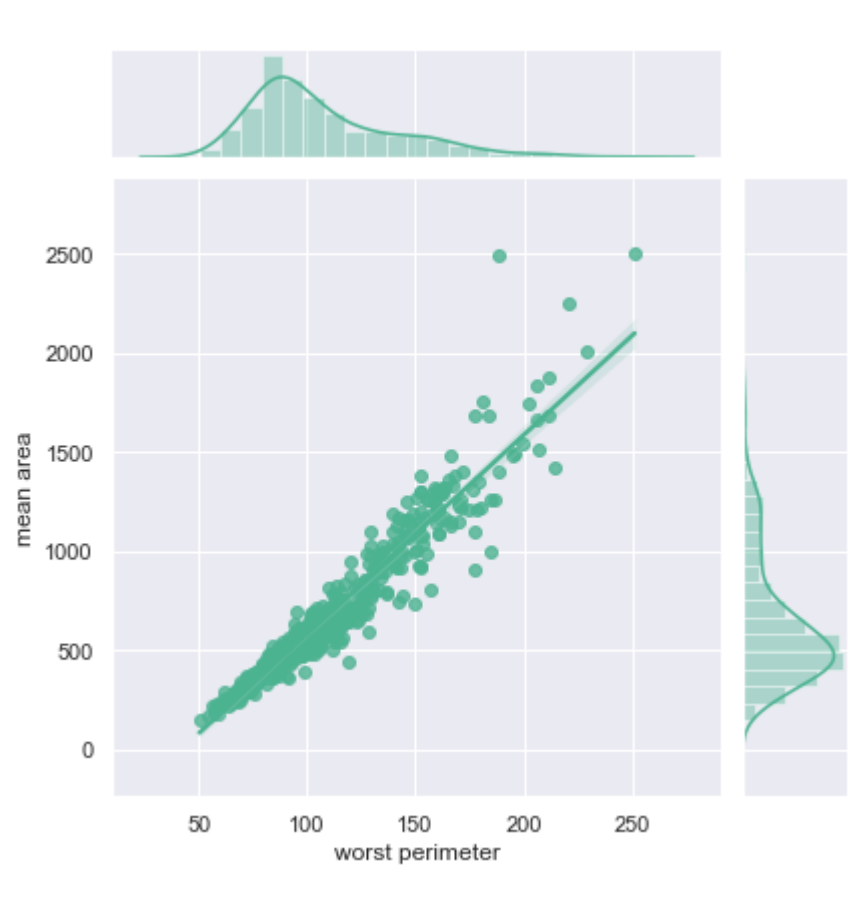
Kde

In [7]:
```
ax1 = sns.jointplot(features['mean radius'], features['mean area'], kind='hex', color='orange')
ax2 = sns.jointplot(features['worst perimeter'], features['worst concave points'], kind='kde', color='#FFAE42')
ax3 = sns.jointplot(features['worst radius'], features['mean radius'], kind='scatter', color='#FFCC00')
ax4 = sns.jointplot(features['worst perimeter'], features['mean area'], kind='reg', color='#4CB391')
```









Again, all these plots confirm feature correlation on a number of levels. We will see a technique later (PCA) on how to remove redundant features.

Prepare the data for the Machine Learning Model

1. One-Hot encoding (no need to perform this for our dataset)
2. Split into class/features
3. Split into train/test
4. Scaling/Standartization if needed

Scaling should be performed after spliting into train/test so that future 'unseen' information does not influence our training set

Split into class/features

In [8]:
```
labels = features['class']
features.drop('class', inplace=True, axis=1)
```

Split into train/test

In [9]:
```
train_features, test_features, train_labels, test_labels = train_test_split(features, labels, test_size=0.2,
                                                                             random_state=1)

# Checkout the shape to confirm everything's good
print("The shape of training features", train_features.shape)
print("The shape of training labels", train_labels.shape)
print("The shape of testing features", test_features.shape)
print("The shape of testing labels", test_labels.shape)
```

```
The shape of training features (455, 30)
The shape of training labels (455,)
The shape of testing features (114, 30)
The shape of testing labels (114,)
```

Feature scaling

We need to perform feature scaling because some measurements (like *fractal dimension error*, check feature plots for a general overview) are in the order of 0.0-0.03 while others (like *worst area*) are 4 orders of magnitude larger! This will inevitably confuse our machine learning method because we will use SVM. SVM uses distance measures and thus this distance measure will be influenced much more by features like *worst area*. This essentially means that other features which are not in the same order of magnitude are effectively rendered useless. To avoid this we perform feature scaling. **However, lets quickly run the algorithm without feature scaling to actually see the benefits first hand.**

First go, without feature scaling: Train and Test the model

In [10]:
```
# Train model
svm = SVC()
svm.fit(train_features, train_labels)

# Test model
predictions = svm.predict(test_features)

print("The accuracy of the model without feature scaling is {} %".format(round(
                                                    100*accuracy_score(test_labels, predictions), 2)))
```

```
The accuracy of the model without feature scaling is 90.35 %
```

Perform feature scaling

Possible Scalers: MinMaxScaler, StandardScaler, RobustScaler, etc. Which one do we choose?

Choosing a scaler depends entirely on the task, data and algorithm. **Only by empirical analysis can we find out which one is better.**

From sklearn.preprocessing:

- MinMaxScaler: Scale data to the range of [0,1]. No distortion on the data. Data is simply being scaled.
- RobustScaler: Dampens the outliers. However, one might want to consider removing the outliers first.
- StandardScaler: subtract the mean and scale to unit variance by dividing by the value of the S.D. However, distorts the relative distance between the feature values. Result is a normal distribution.

We will try out min-max and standard scaler for our model since we have no outliers.

Standard Scaler

In [11]:
```
# Scale training data
scaler = StandardScaler()

# Transform values
train_features2 = pd.DataFrame(scaler.fit_transform(train_features))

# Set columns
train_features2.columns= train_features.columns

# Set index
train_features2.index = train_features.index.values

test_features2 = pd.DataFrame(scaler.fit_transform(test_features))
test_features2.columns = test_features.columns
test_features2.index = test_features.index.values
```

Min-Max

In [12]:
```
scaling = MinMaxScaler(feature_range=(0,1))

train_features3 = pd.DataFrame(scaling.fit_transform(train_features))
train_features3.columns = train_features.columns
train_features3.index = train_features.index.values

test_features3 = pd.DataFrame(scaling.fit_transform(test_features))
test_features3.columns = test_features.columns
test_features3.index = test_features.index.values
```

Train the model

In [13]:
```
svm1 = SVC()
svm2 = SVC()

svm1.fit(train_features2, train_labels);
svm2.fit(train_features3, train_labels);
```

Test the model

In [14]:
```
prediction2 = svm1.predict(test_features2)
prediction3 = svm2.predict(test_features3)

acc1 = 100*accuracy_score(test_labels, predictions)
acc2 = 100*accuracy_score(test_labels, prediction2)
acc3 = 100*accuracy_score(test_labels, prediction3)

print("Accuracy without scaler: {}%".format(round(acc1, 2)))
print("Accuracy with Standard Scaler {}%".format(round(acc2, 2)))
print("Accuracy with Min-Max scaler {}%".format(round(acc3, 2)))
```

```
Accuracy without scaler: 90.35%
Accuracy with Standard Scaler 96.49%
Accuracy with Min-Max scaler 97.37%
```

So the best scaler is the Min-Max scaler in our case.

Evaluate the Model

In [15]:
```
train_features = train_features3
test_features = test_features3

# Confusion Matrix
confusion_mat = confusion_matrix(test_labels, prediction3)

print("Model accuracy:", round(acc3, 3))
print("Micro precision:", round(100*sum(np.diag(confusion_mat))/sum(sum(confusion_mat)), 2))

print("Precission:", round(100*precision_score(test_labels, prediction3), 2))
print("Recall:", round(100*recall_score(test_labels, prediction3), 2))
print("F1 Score:", round(100*f1_score(test_labels, prediction3), 2))
```

```
Model accuracy: 97.368
Micro precision: 97.37
Precission: 97.26
Recall: 98.61
F1 Score: 97.93
```

In [16]:
```
print("Number of samples per class, real data",collections.Counter(np.array(test_labels)))
print("Number of samples per class, predicted data",collections.Counter(prediction3))
```

```
Number of samples per class, real data Counter({1: 72, 0: 42})
Number of samples per class, predicted data Counter({1: 73, 0: 41})
```

Adjusting the Model

Our machine learning looks quite nice:

```
svm = SVC()
svm.fit(train_features, train_labels)
predictions = svm.predict(test_features)
```

3 lines of code to build a model that predicts malignant or begning cancer with *97.37%* accuracy. Well this for once shows how powerful (and wonderful) scikit-learn is. But looking at the first line of code *svm = SVC()* we basically did nothing. So lets take a look at possible hyperparameters that we can tweak.

Generally speaking there are three ways of adjusting a model and making it better.

Sorted according to their efficiency:

1. **Gather more data.** This should be a thumb up rule. In this case we can't get any other data.
2. **Hyperparameter tuning.** This we investigate next.
3. **Using other models**. This we investigate in the other notebook.

Hyperparameter tunning

More info on Support Vector Machines can be found in their the docs.

We will tune the following hyperparameters.

1. **c:** represents the penalty. Controls the trade-off between smooth decision boundary and classifying data points correctly (wiggly to account for all values). A high penalty will cause overfitting (get more training points correctly).
   Basically a higher **C** means a lower regularization which means that the bias is low and the variance is high, that is overfit.
2. **Gamma:** controls how far the influence of single training set reaches.
   - $e^{-\gamma(a-b)^2}$
   - large gamma: close reach (implies overfit)
   - small gamma: far reach

   If Gamma has a very high value, then the decision boundary is just going to be dependent upon the points that are very close to the line which effectively results in ignoring points which are far from the decision boundary.
   Basically a higher γ will lead to overfit.
3. **Kernel:** specifies the kernel type to be used by the algorithm

Using RandomizedSearchCV

docs

Using Sciit-Learn's RandomizedSearchCV method, we can define a grid of hyperparameter ranges and randomly sample from this grid by performing K-Fold CV.

Out of 168 different combinations we will perform only 100 iterations.

In [17]:
```
# 168 different options
random_grid = {
    'C': [0.01, 0.1, 1, 10, 50, 100, 500],
    'gamma': [100, 10, 1, 0.1, 0.01, 0.001],
    'kernel': ['linear', 'poly', 'rbf', 'sigmoid'],
}

svm_temp = SVC()
svm_random = RandomizedSearchCV(estimator=svm_temp, param_distributions=random_grid, n_iter=100,
                                scoring='neg_mean_absolute_error', cv=10, verbose=3, random_state=1,
                                n_jobs=-1)

svm_random.fit(train_features, train_labels);
```

```
Fitting 10 folds for each of 100 candidates, totalling 1000 fits
```

```
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 4 concurrent workers.
[Parallel(n_jobs=-1)]: Done  24 tasks      | elapsed:    1.9s
[Parallel(n_jobs=-1)]: Done 973 tasks      | elapsed:    5.7s
[Parallel(n_jobs=-1)]: Done 1000 out of 1000 | elapsed:    5.8s finished
```

Take a look at the best params

In [18]:
```
svm_random.best_params_
```

Out[18]:
```
{'kernel': 'linear', 'gamma': 0.1, 'C': 10}
```

Using GridSearchCV

Now that we narrowed down the range of hyperparameters we can concentrate our search around the above settings. We can do this with a GridSearchCV, which evaluates all combinations that we define.

In [19]:
```
# Linear kernel does not use the gamma parameter so we can drop it from our grid.
# Lets try to narrow down the value of C
grid_params = {
    'C': [1, 5, 7, 8, 10, 20, 25, 30, 40, 50, 70, 100],
    'kernel': ['linear'],
    'gamma': [0.1]
}

svm_temp = SVC()
svm_grid = GridSearchCV(estimator=svm_temp, param_grid=grid_params, cv=10, n_jobs=-1, verbose=3)
svm_grid.fit(train_features, train_labels);
```

```
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 4 concurrent workers.
```

```
Fitting 10 folds for each of 12 candidates, totalling 120 fits
```

```
[Parallel(n_jobs=-1)]: Done  40 tasks      | elapsed:    0.1s
[Parallel(n_jobs=-1)]: Done 120 out of 120 | elapsed:    0.4s finished
```

In [20]:
```
svm_grid.best_params_
```

Out[20]:
```
{'C': 8, 'gamma': 0.1, 'kernel': 'linear'}
```

We could continue narrowing down until we get dimishing returns (in terms of accuracy) but lets just stop here and evaluate our best model.

In [21]:
```
best_svm = svm_grid.best_estimator_

# Train
best_svm.fit(train_features, train_labels)

# Test
best_fit_prediction = best_svm.predict(test_features)

# Evaluate
print(round(100*accuracy_score(test_labels, best_fit_prediction), 2))
```

```
98.25
```

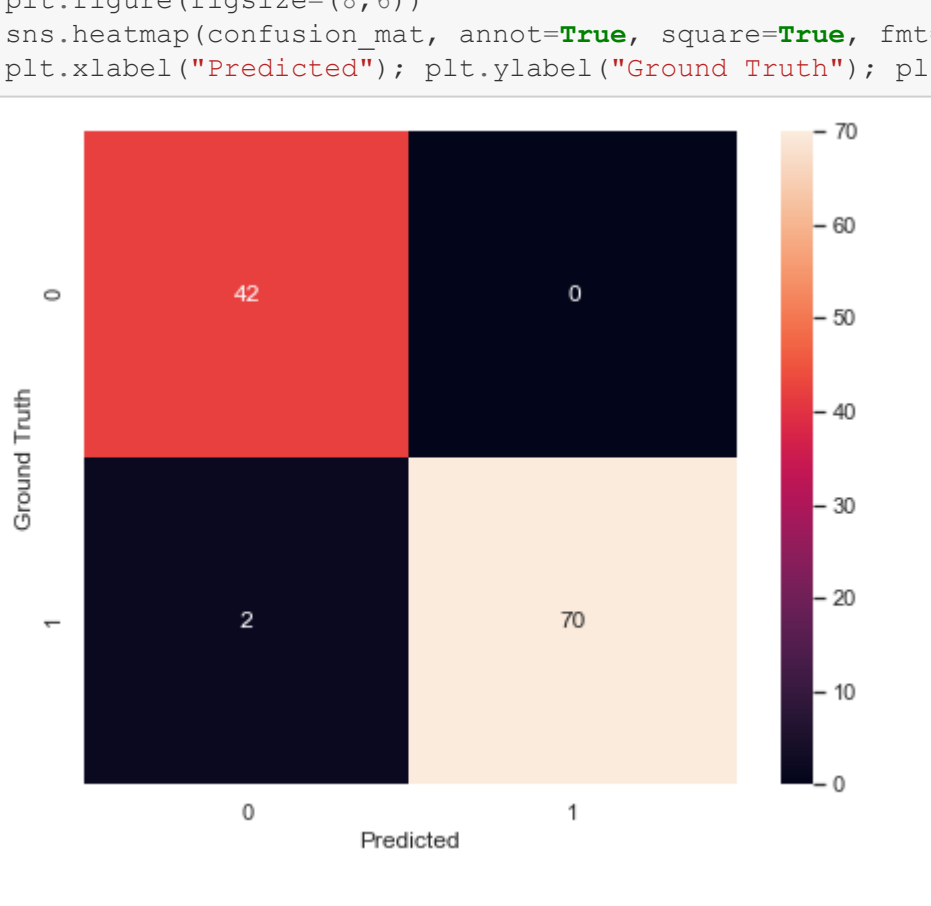
The accuracy got boosted.

In [22]:
```
print("In order to define the hyperplane our model uses {} parameters for the malignant and {} for the \n"
                  "benign class respectively".format(*best_svm.n_support_))
print(best_svm.classes_)
```

```
In order to define the hyperplane our model uses 22 parameters for the malignant and 22 for the
benign class respectively
[0 1]
```

Evaluate the model through the Confusion Matrix

In [23]:
```
confusion_mat = confusion_matrix(test_labels, best_fit_prediction)
plt.figure(figsize=(8,6))
sns.heatmap(confusion_mat, annot=True, square=True, fmt='d', cbar=True)
plt.xlabel("Predicted"); plt.ylabel("Ground Truth"); plt.show()
```



So we can see that we misclassify only 2 samples and these 2 are moreover False Positives, which means that yes we indeed misclassifed two persons to have cancer but in reality they do not have! This is much better when compared to the more insidious False Negative when we wrongly classify a person as not having cancer when he does!

Principal Component Analysis

Docs

Motivation for PCA: Feature importances

One way to check feature importances is through good plots

In [24]:
```python
fig, axes = plt.subplots(10, 3, figsize=(15,9))

malignant = features[labels==0]
benign = features[labels==1]

# flaten axes with numpy ravel
ax = axes.ravel()

for i in range(30):
    _,bins=np.histogram(features.iloc[:,i],bins=50)

    # Alpha defines transperancy in overlapping regions
    ax[i].hist(malignant.iloc[:,i],bins=bins,color='r',alpha=.5)
    ax[i].hist(benign.iloc[:,i],bins=bins,color='g',alpha=.5)
    ax[i].set_title(features.columns[i],fontsize=9)

    # Remove axes
    ax[i].axes.get_xaxis().set_visible(False)
    ax[i].set_yticks(())

ax[0].legend(['malignant','benign'],loc='best',fontsize=8)
plt.tight_layout()
plt.show()
```

We can clearly see that some features like *worst concave points* or *worst area* do a rather good job in separating the classes by themselves.

Other features like *worst symmetry*, *symmetry error* or *mean fractial diension* do not help.

With the help of PCA we can drop some of the features and then possibly improve the performance in this case (although PCA's value does not exactly rely in improving accuracy).

In [25]:
```python
pipeline = Pipeline([('PCA', PCA()), ('SVM', SVC())])

grid_search = {
    'PCA__n_components': [i for i in range(30)],
    'SVM__C': [8],
    'SVM__kernel': ['linear'],
    'SVM__gamma': [0.1]
}

svm_grid = GridSearchCV(estimator=pipeline, param_grid=grid_search, cv=10, n_jobs=-1, verbose=5)
svm_grid.fit(train_features, train_labels);
```

```
Fitting 10 folds for each of 30 candidates, totalling 300 fits
```

```
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 4 concurrent workers.
[Parallel(n_jobs=-1)]: Done  12 tasks      | elapsed:    0.0s
[Parallel(n_jobs=-1)]: Done 272 tasks      | elapsed:    1.1s
[Parallel(n_jobs=-1)]: Done 300 out of 300 | elapsed:    1.2s finished
```

In [26]:
```python
print(svm_grid.best_params_)
predictions = svm_grid.best_estimator_.predict(test_features)
print("Accuracy {}%".format(round(100*accuracy_score(test_labels, predictions),2)))
```

```
{'PCA__n_components': 8, 'SVM__C': 8, 'SVM__gamma': 0.1, 'SVM__kernel': 'linear'}
Accuracy 96.49%
```

Performance did not improve

Conclusions

We went through the data science pipeline of working with data and built a ML model (SVM) in order to tell benign and malignant cancers apart.

Our model is able to predict with **98.25%** accuracy, given certain properties of extracted cancerous cells, whether this cancer is malignant or benign. Our model did even better than the model reported in this paper which achieved **97.1%** accuracy. Just showing again as to how good scikit-learn is. Moreover, the mentioned paper uses RBF as kernel and this is what we also originally used here but through testing we found that a linear kernel performs better. This should be another reason for the success of our model.

Next we investigate two other ML models (RF and LR) in order to see whether we can get a better model.