The CS1109 Dataset
Evaluation Methods for Classification

- Holdout (and many other methods) rely on *randomly* partitioning the dataset into a training set and a test set
  - We've discussed before that the split may be 'lucky' or 'unlucky', hence resampling methods such as *k*-Fold Cross-Validation
- In the case of classification, for example, the split might not reflect the distribution of examples within the classes:
  - Examples of one class might be under-represented in the training set or test set
  - Examples of one class might even be completely absent from the training set or test set
- **Stratification** is the solution to this:
  - In stratification, the proportion of examples of each class in the overall dataset is respected in the partitioning into training and test sets
  - Here's pseudocode for **stratified holdout**. For simplicity, the pseudocode only covers the case of binary classification:
    - Divide the dataset into positive examples, \( P \), and negative examples, \( N \)
    - Randomly partition \( P \) into \( \text{Train}_P \) and \( \text{Test}_P \)
    - Randomly partition \( N \) into \( \text{Train}_N \) and \( \text{Test}_N \)
    - \( \text{Train} \leftarrow \text{Train}_P \cup \text{Train}_N \)
    - \( \text{Test} \leftarrow \text{Test}_P \cup \text{Test}_N \)
- Although this fixes the distribution with respect to the classes, you may still get 'lucky' or 'unlucky' in other ways, so you will still want to do the above multiple times, e.g. **Stratified *k*-Fold Cross-Validation**

Stratification in scikit-learn

In [7]:
```python
# Stratified holdout
ss = StratifiedShuffleSplit(n_splits=1, train_size=0.8)
np.mean(cross_val_score(pipeline, df, y, scoring="accuracy", cv=ss))
```
```
Out[7]: 0.8
```

In [8]:
```python
# Compare the following, which doesn't use stratification, to the above, which does
ss = ShuffleSplit(n_splits=1, train_size=0.8)
np.mean(cross_val_score(pipeline, df, y, scoring="accuracy", cv=ss))
```
```
Out[8]: 0.8285714285714286
```
Performance Measurement for Classification

- We saw before that the performance measure we use to evaluate a predictor might be different from the loss function we used when training it.
  - E.g. for regression we might use MAE or RMSE to measure performance, but (half of) MSE as the loss function.
  The same is true for classification but there is an even greater choice of performance measures.
- The obvious performance measure for a classifier is **accuracy** (used above): the ratio of the number of correct predictions to number of predictions made.
  - If you like notation:

\[
\frac{1}{|T|} \sum_{i=1}^{|T|} I(\hat{y}^{(i)} = y^{(i)})
\]

where \(T\) is the set of examples on which you are testing the classifier and hence \(|T|\) is the number of examples, and \(I(p)\) is the indicator function that outputs 1 if predicate \(p\) is true and zero otherwise.
The Confusion Matrix

- The confusion matrix $CM$ for a classifier is a square $|C| \times |C|$ matrix
  - In a confusion matrix, a cell $CM[i, j]$ contains the number of test examples of class $i$ that were classified as class $j$
- Here are examples of confusion matrices for a binary classifier and a multiclass classifier:

<table>
<thead>
<tr>
<th>Predicted Class</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Actual Class 0</td>
<td>25</td>
<td>10</td>
</tr>
<tr>
<td>Actual Class 1</td>
<td>20</td>
<td>45</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Predicted Class</th>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Actual Class 0</td>
<td>10</td>
<td>0</td>
<td>15</td>
</tr>
<tr>
<td>Actual Class 1</td>
<td>5</td>
<td>30</td>
<td>10</td>
</tr>
<tr>
<td>Actual Class 2</td>
<td>5</td>
<td>5</td>
<td>20</td>
</tr>
</tbody>
</table>

- Let's assume a test set $T$
  - The sum of all entries in $CM$ equals $|T|$
  - The sum of the entries in row $i$ is the number of examples in $T$ that have class $i$
  - The sum of the entries in column $j$ is the number of examples in $T$ that the classifier assigns to class $j$
  - Entries on the main diagonal $CM[i, i]$, are correctly classified, and so $\sum_i CM[i, i]$ is the total number of correctly classified examples.
  - Entries off the main diagonal, $CM[i, j], i \neq j$, are incorrectly classified, and so $\sum_i \sum_{j \neq i} CM[i, j]$ is the total number of incorrectly classified examples.
- So, in a word, what does this calculate?
  $$\frac{\sum_i CM[i, i]}{|T|}$$

Confusion Matrices for Binary Classification

- In the case of binary classification, where we distinguish a positive class from a negative class, there is some special terminology associated with the cells of the confusion matrix:

<table>
<thead>
<tr>
<th>Predicted Class</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Actual Class 0</td>
<td>True Negatives</td>
<td>False Positives</td>
</tr>
<tr>
<td>Actual Class 1</td>
<td>False Negatives</td>
<td>True Positives</td>
</tr>
</tbody>
</table>

- The True Negatives (TN) and True Positives (TP) are correct classifications
- The False Negatives (FN) and False Positives (FP) are incorrect classifications

Confusion Matrices in scikit-learn

In [12]:
```python
y_predicted = cross_val_predict(pipeline, df, y, cv=10) # NB cross_val_predict, not cross_val_score
confusion_matrix(y, y_predicted)
```

Out[12]:
```
array([[179,  27],
       [ 47,  89]])
```
The Limitations of Classification Accuracy

- Accuracy summarizes overall performance in a single figure, which is a good thing
- But it has at least three problems:
  1. Giving only a single figure hides information about how the classifier performs on the individual classes
     - This problem becomes more acute when the costs of different kinds of misclassification are not equal
     - For example, in email classification, it is more serious to misclassify ham as spam
     - Class exercise: Classifier A and classifier B have the same classification accuracy (0.6) but which classifier would you use?

<table>
<thead>
<tr>
<th>Classifier A</th>
<th>Predicted Class</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Benign</td>
</tr>
<tr>
<td>Actual Class</td>
<td></td>
</tr>
<tr>
<td>Benign</td>
<td>400</td>
</tr>
<tr>
<td>Malignant</td>
<td>300</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Classifier B</th>
<th>Predicted Class</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Benign</td>
</tr>
<tr>
<td>Actual Class</td>
<td></td>
</tr>
<tr>
<td>Benign</td>
<td>200</td>
</tr>
<tr>
<td>Malignant</td>
<td>100</td>
</tr>
</tbody>
</table>

- In principle, we could assign costs to the different kinds of mis-classification and define a cost-sensitive variant of classification accuracy, but, in practice, it's difficult, if not impossible, to come up with the costs; for example, how much worse is it to classify ham as spam than spam as ham?
  2. Classification accuracy is also best when the distribution of classes in \( T \) is reasonably balanced
     - If, on the other hand, some of the classes are more prevalent than others, then they tend to bias the measure, e.g. if you do well on the more prevalent classes, then you get a higher score overall
  3. Accuracy does not take into account correct classifications from mere chance
     - There are performance measures that correct for chance (but we won’t look at them!)

Majority-Class Classifier

- Consider \( T \) that contains 950 positive examples and 50 negative examples
- An extremely effective classifier in terms of accuracy for this \( T \) is the so-called majority-class classifier.
  - It always predicts the majority class
  - In this example, it predicts the positive class, and its accuracy is very high: 0.95
  - But it isn’t really a good classifier: it has no ability to discriminate between positive and negative examples
  - Many people compare the accuracy of their classifier(s) against the accuracy of the majority-class classifier to check that they are doing better than this simple-minded baseline
    - It is one way of partly overcoming the second and third problems on the previous slide

```
In [13]: est = DummyClassifier(strategy = "most_frequent")
np.mean(cross_val_score(est, df, y, scoring="accuracy", cv=10))
```

Out[13]: 0.6024242424242425

Exercise

- The majority-class classifier is a ‘baseline’ that we can compare against when evaluating a classifier
- Propose a baseline that you can compare against when evaluating a regressor
- Is it in scikit-learn?
Other Performance Measures for Classification

- If you do any work on classification beyond this module, you'd need to make yourself aware of all the other performance measures that you could use:
  - E.g. precision recall, roc curves, ...
- Beware of 'fishing expeditions':
  - Choose a measure or two in advance of running any experiments — the measure(s) that you think are best-aligned to your business problem
  - When there are many measures, there is a temptation to calculate them all and then get excited when the learner performs well on one of them

Example: the Iris Dataset

- \( m = 150 \) examples: each one is a flower — in fact, an Iris
- \( n = 4 \) features: sepal length, sepal width, petal length and petal width (all in centimetres)
- Three classes (different kinds of Iris): *Iris setosa*, *Iris versicolor* and *Iris virginica*

Big warning:
- The dataset is sorted: 50 setosa then 50 versicolor then 50 virginica
- So shuffling is important

```
In [14]: df = pd.read_csv("datasets/dataset_iris.csv")
       df = df.take(np.random.permutation(len(df)))

In [15]: df.shape
Out[15]: (150, 5)

In [16]: df.columns
Out[16]: Index(["sepal_length", "sepal_width", "petal_length", "petal_width", "label ", dtype='object')

In [17]: df.dtypes
Out[17]: sepal_length    float64
       sepal_width     float64
       petal_length    float64
       petal_width     float64
       label            object
dtype: object
```
In [18]:
df.describe(include="all")

Out[18]:

<table>
<thead>
<tr>
<th></th>
<th>sepal_length</th>
<th>sepal_width</th>
<th>petal_length</th>
<th>petal_width</th>
<th>label</th>
</tr>
</thead>
<tbody>
<tr>
<td>count</td>
<td>150.000000</td>
<td>150.000000</td>
<td>150.000000</td>
<td>150.000000</td>
<td>150</td>
</tr>
<tr>
<td>unique</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>3</td>
</tr>
<tr>
<td>top</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>Iris-versicolor</td>
</tr>
<tr>
<td>freq</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>50</td>
</tr>
<tr>
<td>mean</td>
<td>5.843333</td>
<td>3.054000</td>
<td>3.758667</td>
<td>1.198667</td>
<td>NaN</td>
</tr>
<tr>
<td>std</td>
<td>0.828066</td>
<td>0.433594</td>
<td>1.764420</td>
<td>0.763161</td>
<td>NaN</td>
</tr>
<tr>
<td>min</td>
<td>4.300000</td>
<td>2.000000</td>
<td>1.000000</td>
<td>0.100000</td>
<td>NaN</td>
</tr>
<tr>
<td>25%</td>
<td>5.100000</td>
<td>2.800000</td>
<td>1.600000</td>
<td>0.300000</td>
<td>NaN</td>
</tr>
<tr>
<td>50%</td>
<td>5.800000</td>
<td>3.000000</td>
<td>4.350000</td>
<td>1.300000</td>
<td>NaN</td>
</tr>
<tr>
<td>75%</td>
<td>6.400000</td>
<td>3.300000</td>
<td>5.100000</td>
<td>1.800000</td>
<td>NaN</td>
</tr>
<tr>
<td>max</td>
<td>7.900000</td>
<td>4.400000</td>
<td>6.900000</td>
<td>2.500000</td>
<td>NaN</td>
</tr>
</tbody>
</table>

In [19]:
plengths = df["petal_length"]
pwidths = df["petal_width"]
lables = df["label"]

fig = plt.figure(figsize=(10,5))
plt.title("Petal width against petal length")
plt.scatter(plengths[labels=="Iris-setosa"], pwidths[labels=="Iris-setosa"], color="green")
plt.scatter(plengths[labels=="Iris-versicolor"], pwidths[labels=="Iris-versicolor"], color="red")
plt.scatter(plengths[labels=="Iris-virginica"], pwidths[labels=="Iris-virginica"], color="yellow")
plt.xlabel("Petal length (cm)"")
plt.xlim(0, 8)
plt.ylabel("Petal width (cm)"")
plt.ylim(0, 3)
plt.show()
Any observation you want to make?

For the hell of it, we’ll fit two multinomial logistic regression models:

- One-versus-rest
- Cross entropy

and use a majority-class classifier (although we don’t need to. Why?)

Note: scikit-learn wants the class labels to be integers

In the `CS1109` dataset, they were already 0 or 1

In the Iris dataset, we must convert from “Iris-setosa”, “Iris-versicolor” and “Iris-virginica” to 0, 1, 2

Note: we do not usually one-hot encode class labels (although we will do this when we study neural networks in `CS4619`)

We use `LabelEncoder`

```python
In [20]:
y = df["label"].values
encoder = LabelEncoder()
y_encoded = encoder.fit_transform(y)
```

```python
In [21]:
# The features we want to select
features = ["sepal_length", "sepal_width", "petal_length", "petal_width"]

# Create the one-versus-rest pipeline
ovr_pipeline = Pipeline([  
    ("pre", ColumnTransformer([("num", StandardScaler(), features)], remainder="drop")),  
    ("est", LogisticRegression())
])

# Create the cross-entropy pipeline
cross_pipeline = Pipeline([  
    ("pre", ColumnTransformer([("num", StandardScaler(), features)], remainder="drop")),  
    ("est", LogisticRegression(multi_class="multinomial", solver="newton-cg"))
])

# Create the majority-class classifier
maj_pipeline = DummyClassifier(strategy = "most_frequent")
```

```python
In [22]:
print("One-versus-Rest: ", np.mean(cross_val_score(ovr_pipeline, df, y_encoded, scoring="accuracy", cv=10)))
print("Cross entropy: ", np.mean(cross_val_score(cross_pipeline, df, y_encoded, scoring="accuracy", cv=10)))
print("Majority-class classifier: ", np.mean(cross_val_score(maj_pipeline, df, y_encoded, scoring="accuracy", cv=10)))
```

```
One-versus-Rest:  0.9066666666666666
Cross entropy:  0.9533333333333334
Majority-class classifier:  0.3333333333333337
```