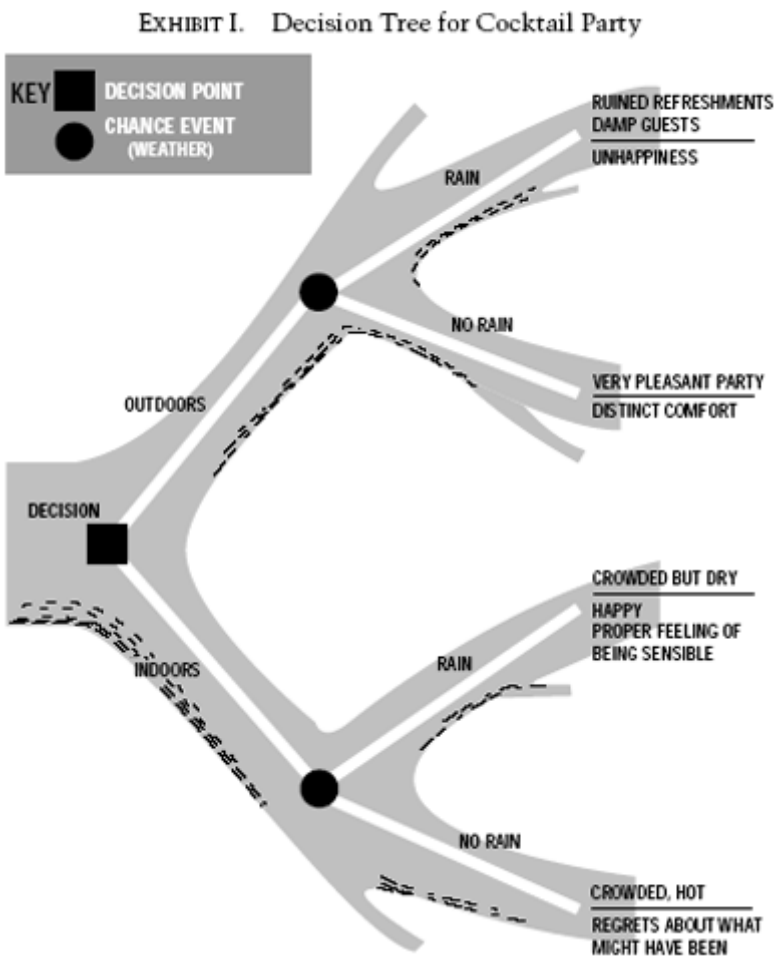
Introduction

A decision tree is essentially a series of if-then statements, that, when applied to a record in a data set, results in the classification of that record. Therefore, once you've created your decision tree, you will be able to run a data set through the program and get a classification for each individual record within the data set. What this means to you, as a manufacturer of quality widgets, is that the program you create from this article will be able to predict the likelihood of each user, within a data set, purchasing your finely crafted product.

Decision tree is a type of supervised learning algorithm (having a pre-defined target variable) that is mostly used in classification problems. It works for both categorical and continuous input and output variables. In this technique, we split the population or sample into two or more homogeneous sets (or sub-populations) based on most significant splitter / differentiator in input variables.



How decision tree works

The understanding level of Decision Trees algorithm is so easy compared with other classification algorithms. The decision tree algorithm tries to solve the problem, by using tree representation. Each internal node of the tree corresponds to an attribute, and each leaf node corresponds to a class label.

Decision Tree Algorithm Pseudocode

- Place the best attribute of the dataset at the root of the tree.
- Split the training set into subsets. Subsets should be made in such a way that each subset contains data with the same value for an attribute.
- Repeat step 1 and step 2 on each subset until you find leaf nodes in all the branches of the tree.

Decision Tree classifier

In decision trees, for predicting a class label for a record we start from the root of the tree. We compare the values of the root attribute with record's attribute. On the basis of comparison, we follow the branch corresponding to that value and jump to the next node.

We continue comparing our record's attribute values with other internal nodes of the tree until we reach a leaf node with predicted class value. As we know how the modeled decision tree can be used to predict the target class or the value. Now let's understanding how we can create the decision tree model.

Assumptions while creating Decision Tree

The below are the some of the assumptions we make while using Decision tree:

- At the beginning, the whole training set is considered as the root.
- Feature values are preferred to be categorical. If the values are continuous then they are discretized prior to building the model.
- Records are distributed recursively on the basis of attribute values.
- Order to placing attributes as root or internal node of the tree is done by using some statistical approach.

How to split nodes

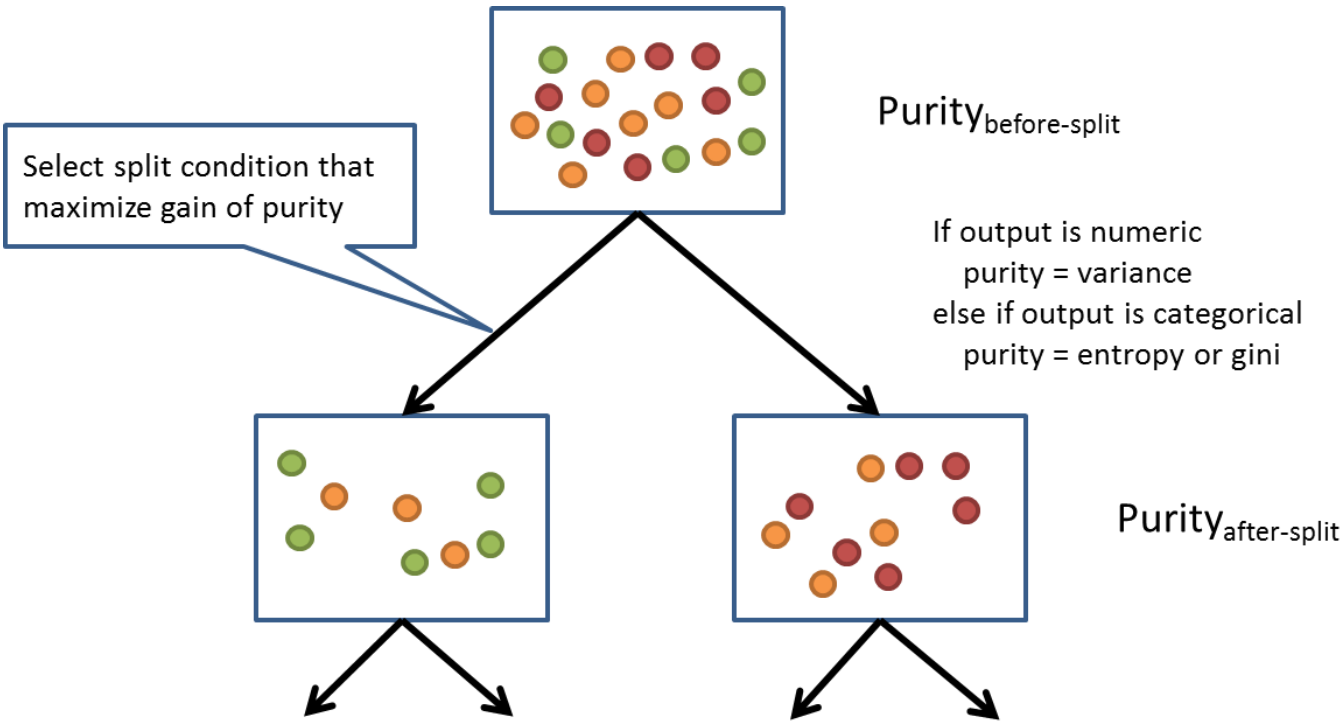
There are few algorithms to find optimum split. Let's look at the following to understand the mathematics behind.

Entropy

An alternative splitting criterion for decision tree learning algorithms is *information gain*. It measures how well a particular attribute distinguishes among different target classifications. Information gain is measured in terms of the expected reduction in the entropy or impurity of the data. The entropy of a set of probabilities is:

$$H(p) = -\sum_i p_i log_2(p_i)$$

If we have a set of binary responses from some variable, all of which are positive/true/1, then knowing the values of the variable does not hold any predictive value for us, since all the outcomes are positive. Hence, the entropy is zero:



The entropy calculation tells us how much additional information we would obtain with knowledge of the variable.

So, if we have a set of candidate covariates from which to choose as a node in a decision tree, we should choose the one that gives us the most information about the response variable (*i.e.* the one with the highest entropy).

Misclassification Rate

Alternatively, we can use the misclassification rate:

$$C(j,t) = \frac{1}{n_{jt}} \sum_{y_i : x_{ij} > t} I(y_i \neq \hat{y})$$

where $\hat{y}$ is the most probable class label and n_{ij} is the number of observations in the data subset obtained from splitting via j, t.

Gini index

The Gini index is simply the expected error rate:

$$C(j,t) = \sum_{k=1}^{K} \hat{\pi}_{jt}[k](1 - \hat{\pi}_{jt}[k]) = 1 - \sum_{k=1}^{K} \hat{\pi}_{jt}[k]^2$$

where $\hat{\pi}_{jt}[k]$ is the probability of an observation being correctly classified as class k for the data subset obtained from splitting via j, t (hence, $(1 - \hat{\pi}_{jt}[k])$ is the misclassification probability).

ID3

A given cost function can be used to construct a decision tree via one of several algorithms. The Iterative Dichotomiser 3 (ID3) is on such algorithm, which uses entropy, and a related concept, *information gain*, to choose features and partitions at each classification step in the tree.

Information gain is the difference between the current entropy of a system and the entropy measured after a feature is chosen. If S is a set of examples and X is a possible feature on which to partition the examples, then:

$$G(S, X) = \text{Entropy}(S) - \sum_{x \in X} \frac{\#(S_x)}{\#(S)} \text{Entropy}(S_x)$$

where $\#$ is the count function and x is a particular value of X.

Let's say S is a set of survival events, $S = \{s_1 = survived, s_2 = died, s_3 = died, s_4 = died\}$ and a particular variable X can have values $\{x_1, x_2, x_3\}$. To perform a sample calculation of information gain, we will say that:

- $X(s_1) = x_2$
- $X(s_2) = x_2$
- $X(s_3) = x_3$
- $X(s_4) = x_1$

The current entropy of this state is:

$$\begin{aligned} \text{Entropy}(S) &= -p^{(+)} \log_2(p^{(+)}) - p^{(-)} \log_2(p^{(-)}) \\ &= -0.25 \log_2(0.25) - 0.75 \log_2(0.75) \\ &= 0.5 + 0.311 = 0.811 \end{aligned}$$

Now, we need to compute the information after selecting variable X, which is the sum of three terms:

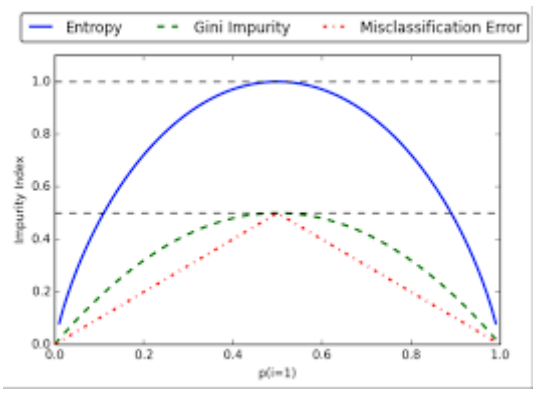
$$\frac{\#(S_{x1})}{\#(S)} \text{Entropy}(S) = 0.25(-0 \log_2(0) - 1 \log_2(1)) = 0$$

$$\frac{\#(S_{x2})}{\#(S)} \text{Entropy}(S) = 0.5(-0.5 \log_2(0.5) - 0.5 \log_2(0.5) = 0.5$$

$$\frac{\#(S_{x3})}{\#(S)} \text{Entropy}(S) = 0.25(-0 \log_2(0) - 1 \log_2 1) = 0$$

Therefore, the information gain is:

$$G(S, X) = 0.811 - (0 + 0.5 + 0) = 0.311$$



Implementation using scikit-learn

In [1]:

```
import numpy as np, pandas as pd, matplotlib.pyplot as plt, pydotplus
from sklearn import tree, metrics, model_selection, preprocessing
from IPython.display import Image, display
```

Load and prep the data

In [2]:

```
# load the iris data
df = pd.read_csv('iris.csv')
df['species_label'], _ = pd.factorize(df['species'])
df.head()
```

Out[2]:

	sepal_length	sepal_width	petal_length	petal_width	species	species_label
0	5.1	3.5	1.4	0.2	Iris-setosa	0
1	4.9	3.0	1.4	0.2	Iris-setosa	0
2	4.7	3.2	1.3	0.2	Iris-setosa	0
3	4.6	3.1	1.5	0.2	Iris-setosa	0
4	5.0	3.6	1.4	0.2	Iris-setosa	0

In [3]:

```
# select features
y = df['species_label']
X = df[['petal_length', 'petal_width']]
```

In [4]:

```
# split data randomly into 70% training and 30% test
X_train, X_test, y_train, y_test = model_selection.train_test_split(X, y, test_size=0.3
, random_state=0)
```

Train the model and make predictions

Note we didn't have to standardize the data to use a decision tree.

In [5]:

```
# train the decision tree
dtree = tree.DecisionTreeClassifier(criterion='entropy', max_depth=3, random_state=0)
dtree.fit(X_train, y_train)
```

Out[5]:

```
DecisionTreeClassifier(class_weight=None, criterion='entropy', max_depth=
3,
            max_features=None, max_leaf_nodes=None,
            min_impurity_split=1e-07, min_samples_leaf=1,
            min_samples_split=2, min_weight_fraction_leaf=0.0,
            presort=False, random_state=0, splitter='best')
```

In [6]:

```
# use the model to make predictions with the test data
y_pred = dtree.predict(X_test)
```

Evaluate the model's performance

Including the tree's axis-parallel decision boundaries and how the tree splits

In [7]:

```
# how did our model perform?
count_misclassified = (y_test != y_pred).sum()
print('Misclassified samples: {}'.format(count_misclassified))
accuracy = metrics.accuracy_score(y_test, y_pred)
print('Accuracy: {:.2f}'.format(accuracy))
```

```
Misclassified samples: 1
Accuracy: 0.98
```

Visualization

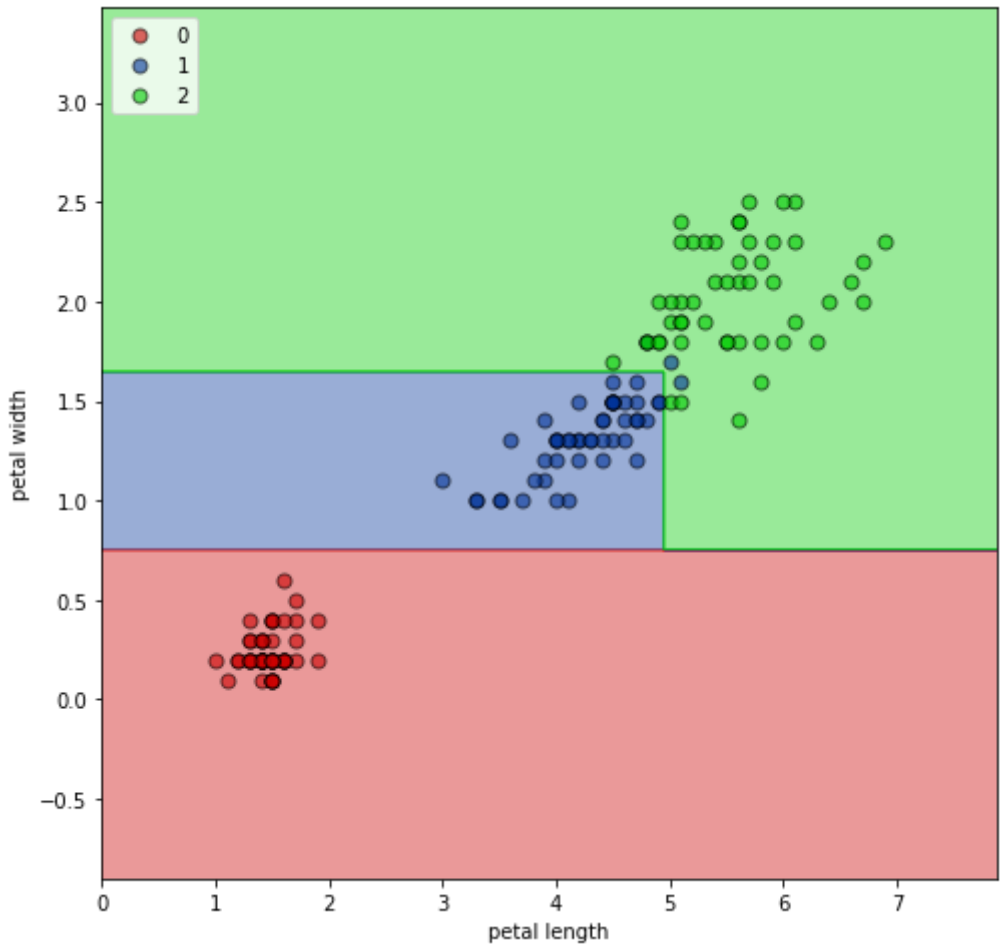
For visualizing decision tree splits I am creating **plot_decision()** function below using matplotlib. If you dont understand the implementation completely that's fine. It is just for the understanding.

In [8]:

```
from matplotlib.colors import ListedColormap


def plot_decision(X, y, classifier, test_idx=None, resolution=0.02, figsize=(8,8)):

    # setup marker generator and color map
    markers = ('s', 'x', 'o', '^', 'v')
    colors = ('#cc0000', '#003399', '#00cc00', '#999999', '#66ffff')
    cmap = ListedColormap(colors[:len(np.unique(y))])

    # get dimensions
    x1_min, x1_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    x2_min, x2_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx1, xx2 = np.meshgrid(np.arange(x1_min, x1_max, resolution), np.arange(x2_min, x2_
max, resolution))
    xmin = xx1.min()
    xmax = xx1.max()
    ymin = xx2.min()
    ymax = xx2.max()

    # create the figure
    fig, ax = plt.subplots(figsize=figsize)
    ax.set_xlim(xmin, xmax)
    ax.set_ylim(ymin, ymax)

    # plot the decision surface
    Z = classifier.predict(np.array([xx1.ravel(), xx2.ravel()]).T)
    Z = Z.reshape(xx1.shape)
    ax.contourf(xx1, xx2, Z, alpha=0.4, cmap=cmap, zorder=1)

    # plot all samples
    for idx, cl in enumerate(np.unique(y)):
        ax.scatter(x=X[y == cl, 0],
                   y=X[y == cl, 1],
                   alpha=0.6,
                   c=cmap(idx),
                   edgecolor='black',
                   marker='o',#markers[idx],
                   s=50,
                   label=cl,
                   zorder=3)

    # highlight test samples
    if test_idx:
        X_test, y_test = X[test_idx, :], y[test_idx]
        ax.scatter(X_test[:, 0],
                   X_test[:, 1],
                   c='w',
                   alpha=1.0,
                   edgecolor='black',
                   linewidths=1,
                   marker='o',
                   s=150,
                   label='test set',
                   zorder=2)
```

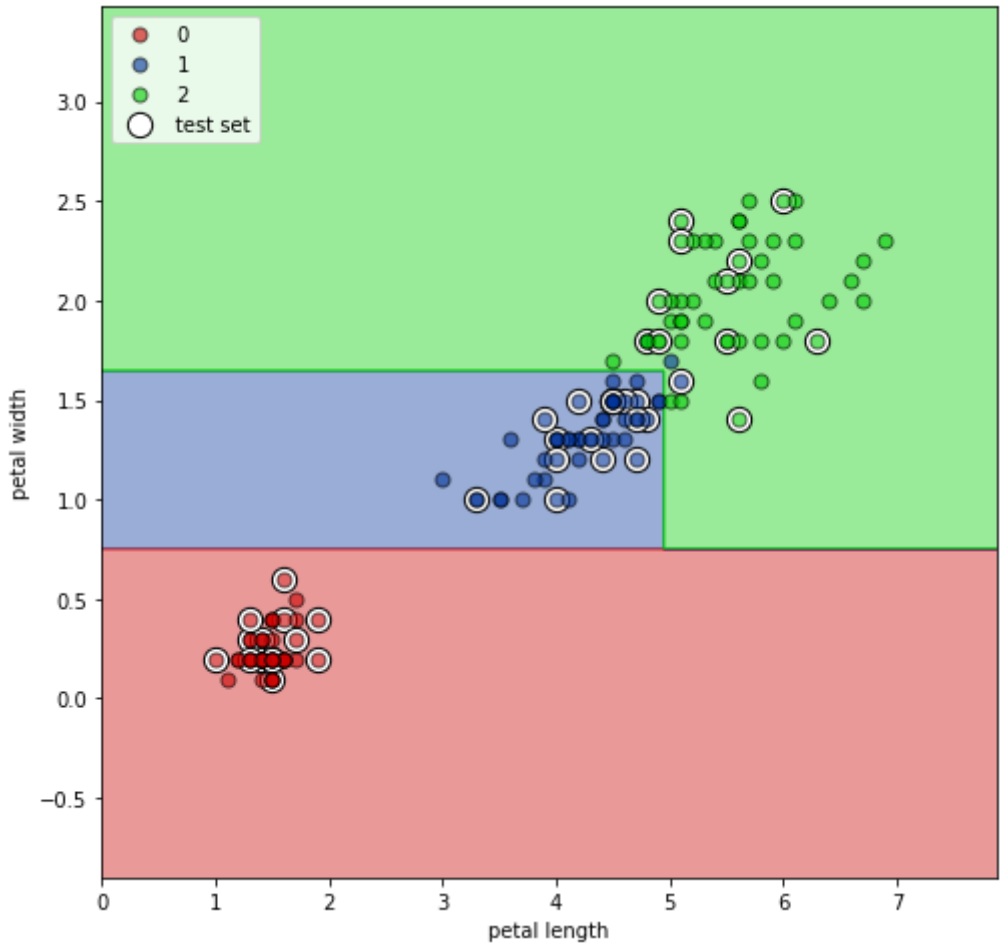
In [9]:

```
# visualize the model's decision regions to see how it separates the samples
X_combined = np.vstack((X_train, X_test))
y_combined = np.hstack((y_train, y_test))
plot_decision(X=X_combined, y=y_combined, classifier=dtree)
plt.xlabel('petal length')
plt.ylabel('petal width')
plt.legend(loc='upper left')
plt.show()
```



In [10]:

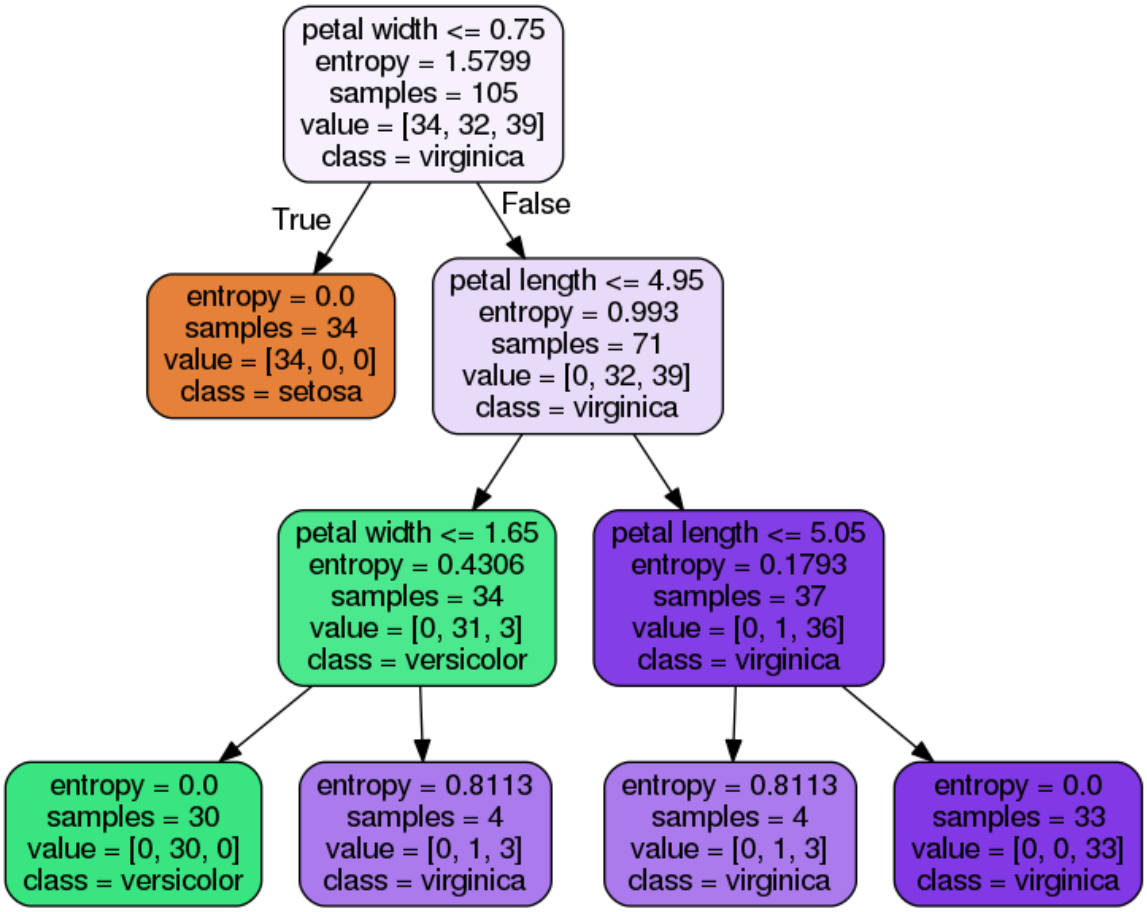
```
# same thing, but this time identify the points that constituted the test data set
test_idx = range(len(y_train), len(y_combined))
plot_decision(X=X_combined, y=y_combined, classifier=dtree, test_idx=test_idx)
plt.xlabel('petal length')
plt.ylabel('petal width')
plt.legend(loc='upper left')
plt.show()
```



Now visualize how the tree splits using GraphViz (http://www.graphviz.org/) (make sure you install it first):

In [11]:

```
dot_data = tree.export_graphviz(dtree, out_file=None, filled=True, rounded=True,
                                feature_names=['petal length', 'petal width'],
                                class_names=['setosa', 'versicolor', 'virginica'])
graph = pydotplus.graph_from_dot_data(dot_data)
display(Image(graph.create_png()))
```

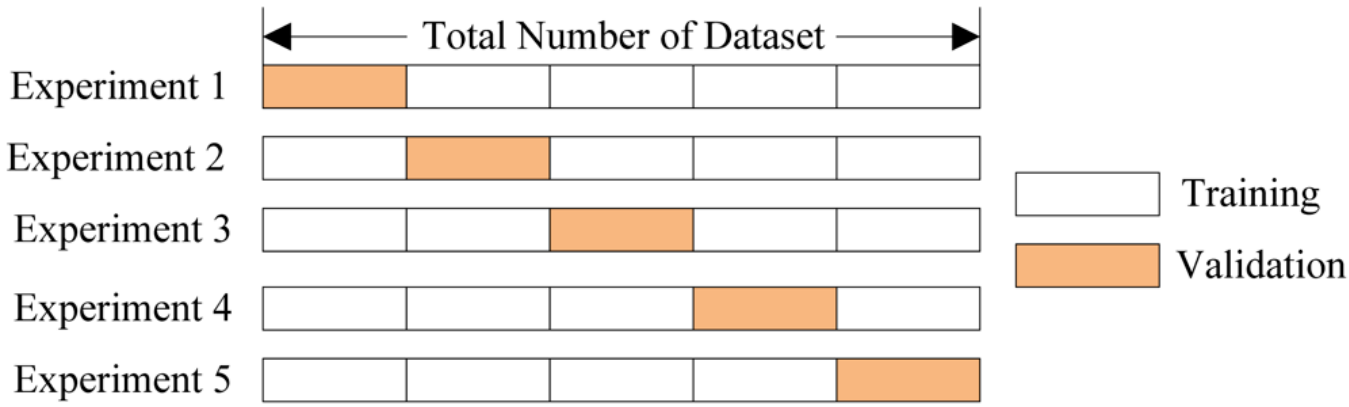


Cross Validation

Cross Validation is a technique which involves reserving a particular sample of a data set on which you do not train the model. Later, you test the model on this sample before finalizing the model.

Here are the steps involved in cross validation:

- You reserve a sample data set.
- Train the model using the remaining part of the data set.
- Use the reserve sample of the data set test (validation) set. This will help you to know the effectiveness of model performance. It your model delivers a positive result on validation data, go ahead with current model. It rocks!



Holdout Validation and Cross Validation

When creating a predictive model, we'd like to get an accurate sense of its ability to generalize to unseen data before actually going out and using it on unseen data. As we saw earlier, generating predictions on the training data itself to check the model's accuracy does not work very well: a complex model may fit the training data extremely closely but fail to generalize to new, unseen data. We can get a better sense of a model's expected performance on unseen data by setting a portion of our training data aside when creating a model, and then using that set aside data to evaluate the model's performance. This technique of setting aside some of the training data to assess a model's ability to generalize is known as validation.

Holdout validation and cross validation are two common methods for assessing a model before using it on test data. Holdout validation involves splitting the training data into two parts, a training set and a validation set, building a model with the training set and then assessing performance with the validation set. In theory, model performance on the hold-out validation set should roughly mirror the performance you'd expect to see on unseen test data. In practice, holdout validation is fast and it can work well, especially on large data sets, but it has some pitfalls.

Reserving a portion of the training data for a holdout set means you aren't using all the data at your disposal to build your model in the validation phase. This can lead to suboptimal performance, especially in situations where you don't have much data to work with. In addition, if you use the same holdout validation set to assess too many different models, you may end up finding a model that fits the validation set well due to chance that won't necessarily generalize well to unseen data. Despite these shortcomings, it is worth learning how to use a holdout validation set in Python.

In [12]:

```
from sklearn.cross_validation import KFold

cv = KFold(n=len(X),  # Number of elements
           n_folds=10,            # Desired number of cv folds
           random_state=12)
```

```
/usr/local/lib/python3.5/dist-packages/sklearn/cross_validation.py:44: Dep
recationWarning: This module was deprecated in version 0.18 in favor of th
e model_selection module into which all the refactored classes and functio
ns are moved. Also note that the interface of the new CV iterators are dif
ferent from that of this module. This module will be removed in 0.20.
  "This module will be removed in 0.20.", DeprecationWarning)
```

After creating a cross validation object, you can loop over each fold and train and evaluate a your model on each one:

In [13]:

```
fold_accuracy = []

# titanic_train["Sex"] = encoded_sex

for train_fold, valid_fold in cv:
    train = X.loc[train_fold] # Extract train data with cv indices
    valid = X.loc[valid_fold] # Extract valid data with cv indices

    train_y = y.loc[train_fold]
    valid_y = y.loc[valid_fold]

    model = dtree.fit(X = train,
                          y = train_y)
    valid_acc = model.score(X = valid,
                           y = valid_y)
    fold_accuracy.append(valid_acc)

print("Accuracy per fold: ", fold_accuracy, "\n")
print("Average accuracy: ", sum(fold_accuracy)/len(fold_accuracy))
```

```
Accuracy per fold:  [1.0, 1.0, 1.0, 1.0, 0.93333333333333335, 0.8000000000
0000004, 1.0, 0.8666666666666667, 0.80000000000000004, 1.0]

Average accuracy:  0.94
```

Model accuracy can vary significantly from one fold to the next, especially with small data sets, but the average accuracy across the folds gives you an idea of how the model might perform on unseen data. As with holdout validation, we'd like the target variable's classes to have roughly the same proportion across each fold when performing cross validation for a classification problem. To perform stratified cross validation, use the StratifiedKFold() function instead of KFold(). You use can score a model with stratified cross validation with a single function call with the cross_val_score() function:

In [14]:

```
from sklearn.cross_validation import cross_val_score
```

In [15]:

```
scores = cross_val_score(estimator= dtree,     # Model to test
               X= X,
               y = y,      # Target variable
               scoring = "accuracy",              # Scoring metric
               cv=10)                             # Cross validation folds

print("Accuracy per fold: ")
print(scores)
print("Average accuracy: ", scores.mean())
```

```
Accuracy per fold:
[ 1.          0.93333333  1.          0.93333333  0.93333333  0.93333333
  0.93333333  0.93333333  1.          1.        ]
Average accuracy:  0.96
```

Overfitting

Overfitting is a practical problem while building a decision tree model. The model is having an issue of overfitting is considered when the algorithm continues to go deeper and deeper in the to reduce the training set error but results with an increased test set error i.e, Accuracy of prediction for our model goes down. It generally happens when it builds many branches due to outliers and irregularities in data.

Two approaches which we can use to avoid overfitting are:

- Pre-Pruning
- Post-Pruning

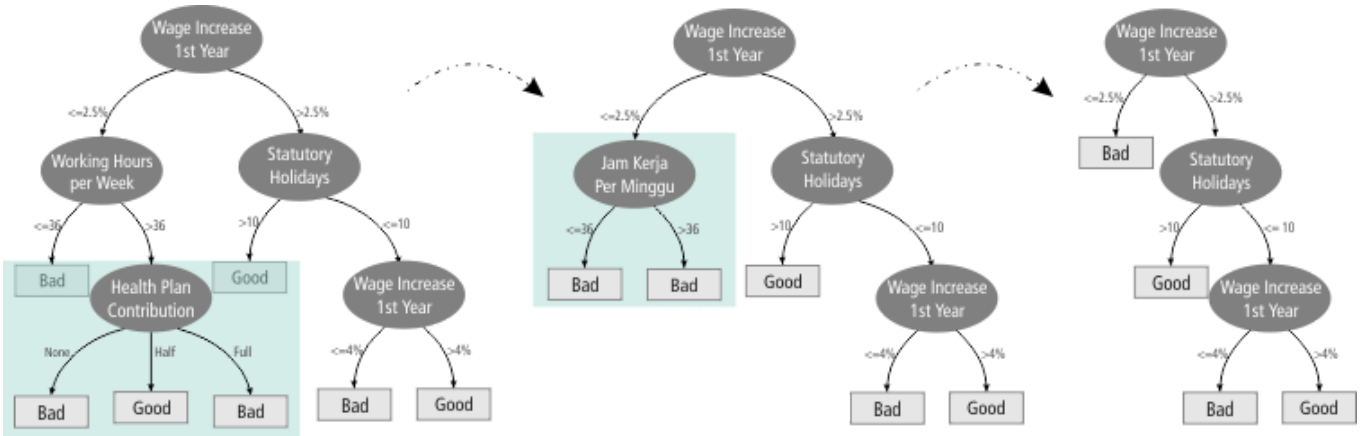
Pre-Pruning

In pre-pruning, it stops the tree construction bit early. It is preferred not to split a node if its goodness measure is below a threshold value. But it's difficult to choose an appropriate stopping point.

Post-Pruning

In post-pruning first, it goes deeper and deeper in the tree to build a complete tree. If the tree shows the overfitting problem then pruning is done as a post-pruning step. We use a cross-validation data to check the effect of our pruning. Using cross-validation data, it tests whether expanding a node will make an improvement or not.

If it shows an improvement, then we can continue by expanding that node. But if it shows a reduction in accuracy then it should not be expanded i.e, the node should be converted to a leaf node.



Decision Tree Algorithm Advantages and Disadvantages

Advantages:

- Decision Trees are easy to explain. It results in a set of rules.
- It follows the same approach as humans generally follow while making decisions.
- Interpretation of a complex Decision Tree model can be simplified by its visualizations. Even a naive person can understand logic.
- The Number of hyper-parameters to be tuned is almost null.

Disadvantages:

- There is a high probability of overfitting in Decision Tree.

- Generally, it gives low prediction accuracy for a dataset as compared to other machine learning algorithms.
- Information gain in a decision tree with categorical variables gives a biased response for attributes with greater no. of categories.
- Calculations can become complex when there are many class labels.

In [ ]: