# DecisionTree

April 3, 2019

```
In [1]: from collections import Counter

        import numpy as np
        from numpy import genfromtxt
        import pandas as pd
        import scipy.io
        from scipy import stats
        import matplotlib.pyplot as plt


        import random

In [2]: class DecisionTree:

            def __init__(self, max_depth=4, prune=True, threshold=.1, shuffle_features=False, n

                self.left = None
                self.right = None
                self.label = None
                self.best_feature = None
                self.best_split_point = None
                self.best_gain = None
                self.n_features = None

                self.n_random_features = n_random_features
                self.shuffle_features = shuffle_features
                self.threshold = threshold
                self.prune = prune
                self.max_depth = max_depth



            @staticmethod
            def gini_impurity(y):

                counter = Counter(y)
                impurity = 1 - sum([(counter[k] / sum(counter.values()))**2 for k in counter.ke
```

```python
        return impurity


    def _split_point(self, X, y):

        best_feature = None
        best_split_point = None
        best_gain = 0

        self.n_features = X.shape[1]

        features = np.arange(self.n_features)

        # for random forest
        if self.shuffle_features:

            if self.n_random_features is None:

                self.n_random_features = int(np.sqrt(self.n_features))

            features = np.random.choice(features, self.n_random_features, replace=Fals


        for col in features:

            for split_point in np.unique(X[:, col]):

                current_impurity = self.gini_impurity(y)

                left_y = y[X[:, col] < split_point]
                right_y = y[X[:, col] >= split_point]

                left_gini = self.gini_impurity(left_y)
                right_gini = self.gini_impurity(right_y)

                left_weight = left_y.shape[0] / X.shape[0]
                right_weight = right_y.shape[0] / X.shape[0]

                impurity = left_weight * left_gini + right_gini * right_weight

                # Compare current impurity with child nodes.
                # Higher gain => Better
                gain = current_impurity - impurity

                if gain > best_gain:
                    best_gain = gain
                    best_feature = col
                    best_split_point = split_point
```

```python
        self.best_gain = best_gain
        self.best_feature = best_feature
        self.best_split_point = best_split_point



    def fit(self, X, y):

        self._fit(X, y)

        if self.prune:
            self.prune_tree(y)



    def _fit(self, X, y):

        if self.max_depth > 0:

            self._split_point(X, y)

            # No split is done -> splitting doesn't produce higher accuracy
            if self.best_feature is None:

                self.label = self._get_label(y)
                return


            left_X = X[X[:, self.best_feature] < self.best_split_point, :]
            left_y = y[X[:, self.best_feature] < self.best_split_point]

            right_X = X[X[:, self.best_feature] >= self.best_split_point, :]
            right_y = y[X[:, self.best_feature] >= self.best_split_point]

            # if one node has all or no points, no split is needed
            if len(left_y) == 0 or len(right_y) == 0:

                self.label = self._get_label(y)
                self.best_feature = None
                self.best_split_point = None

            else:

                self.left = DecisionTree(max_depth=self.max_depth-1, prune=self.prune,
                                             n_random_features=self.n_random_features,

                self.right = DecisionTree(max_depth=self.max_depth-1, prune=self.prune
```

```
                                                    n_random_features=self.n_random_features

                self.left._fit(left_X, left_y)
                self.right._fit(right_X, right_y)


        # current node reached the maximum depth
        else:

            self.label = self._get_label(y)


    def _get_label(self, y):
        counter = Counter(y)
        return int(max(counter.items(), key=lambda x: x[1])[0])


    def predict(self, X):

        return np.array([self._predict(x) for x in X])


    def _predict(self, x):

        # leaf nodes
        if self.max_depth == 0 or self.best_feature is None:

            return self.label

        else:

            if self.left is None or self.right is None:

                return self.label


            if x[self.best_feature] < self.best_split_point:

                return self.left._predict(x)

            return self.right._predict(x)



    def cross_entropy(self, y, pred):

        p = sum((pred==1)) / len(pred)
```

```
        return -(y*np.log(p) + (1-y)*np.log(1-p))


    def prune_tree(self, y):

        # base case : leaf node
        if self.best_feature is None:
            return


        if self.left is not None:
            self.left.prune_tree(y)

        if self.right is not None:
            self.right.prune_tree(y)

        if self.best_gain < self.threshold:
            self.left = self.right = None
            self.best_feature = self.best_split_point = self.impurity = None
            self.label = self._get_label(y)



    def loss(self, y, pred):

        return self.cross_entropy(y, pred).sum() / len(y)


    def accuracy(self, X, y):

        pred = self.predict(X)
        return sum(y==pred) / len(y)



    def __repr__(self):

        return self._show_tree(0)

    def _show_tree(self, d):

        if self.max_depth >= 0:

            indent = '   ' * d

            tree = f'\n{indent}Depth : {d}, Split Feature : {self.best_feature}, Split
            left = right = label = ''
```

```
            if self.left is not None:
                left = self.left._show_tree(d+1)

            if self.right is not None:
                right = self.right._show_tree(d+1)

            if self.left is None and self.right is None:
                return f'\n{indent}Depth : {d}, Label : {self.label}'

            else:
                return tree + left + right




    class RandomForest():

        def __init__(self, max_depth=4, n_estimators=5, threshold=.02, prune=False, n_rand

            self.trees = []

            self.n_estimators = n_estimators
            self.threshold = threshold
            self.prune = prune
            self.max_depth = max_depth
            self.n_random_features = n_random_features
            self.bootstrap = bootstrap

        def fit(self, X, y):

            if self.n_random_features is None:
                self.n_random_features = int(np.sqrt(X.shape[1]))

            for i in range(self.n_estimators):

                tree = DecisionTree(max_depth=self.max_depth, prune=self.prune, threshold=s
                                    shuffle_features=True, n_random_features=self.n_

                if self.bootstrap:
                    idx = np.random.choice(np.arange(X.shape[0]), int(X.shape[0]*.8))
                    data, label = X[idx], y[idx]

                    tree.fit(data, label)
                else:
                    tree.fit(X, y)
                self.trees.append(tree)
```

```
    def predict(self, X):

        pred = []

        for tree in self.trees:
            pred.append(tree.predict(X))

        pred = np.mean(pred, axis=0)

        # if higher than 0.5, it means there were more label 1 than 0.
        pred[pred >= 0.5] = 1
        pred[pred < 0.5] = 0

        return pred

    def accuracy(self, X, y):

        pred = self.predict(X)

        return sum(pred == y) / len(y)
```

```
In [3]: def split(X, size):

            if type(size) == float:
                size = round(len(X) * size)

            dat = X.copy()

            # for reproducibility
            np.random.seed(24)

            # shuffle copied data
            np.random.shuffle(dat)

            # training_data, validation_data
            return dat[size:], dat[:size]

        def fillna(cols):

            for col in cols:

                data[col] = round(data[[col]].fillna(int(round(np.mean(data[col])))))
```

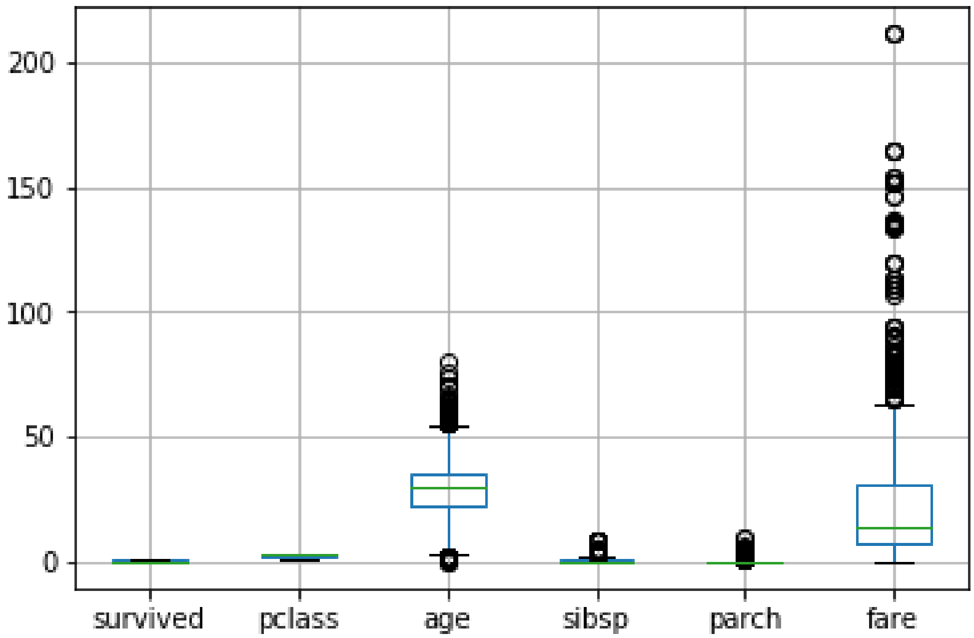
```
In [4]: dataset = "titanic"

        if dataset == "titanic":
            # Load titanic data
            path_train = 'datasets/titanic/titanic_training.csv'
```

```
        data = pd.read_csv(path_train)
        path_test = 'datasets/titanic/titanic_testing_data.csv'
        test_data = pd.read_csv(path_test)
        y = data['survived']
        class_names = ["Died", "Survived"]

    elif dataset == "spam":
        features = [
            "pain", "private", "bank", "money", "drug", "spam", "prescription",
            "creative", "height", "featured", "differ", "width", "other",
            "energy", "business", "message", "volumes", "revision", "path",
            "meter", "memo", "planning", "pleased", "record", "out",
            "semicolon", "dollar", "sharp", "exclamation", "parenthesis",
            "square_bracket", "ampersand"
        ]
        assert len(features) == 32

        # Load spam data
        path_train = 'datasets/spam-dataset/spam_data.mat'
        data = scipy.io.loadmat(path_train)
        X = data['training_data']
        y = np.squeeze(data['training_labels'])
        Z = data['test_data']
        class_names = ["Ham", "Spam"]

    else:
        raise NotImplementedError("Dataset %s not handled" % dataset)
```

#### 0.0.1 Titanic

```
In [5]: fillna(['age', 'sibsp', 'parch'])

        # all col values are NaN
        data = data.drop(index=705).drop(['cabin', 'ticket'], axis=1)

        # index=38. Missing fare with pclass=3 and embakred at S
        data.loc[data['fare'].isna(), 'fare'] = data[(data['embarked'] == 'S') & (data['pclass

        fare_by_embarked = data[data['pclass']==1][['fare', 'embarked']].groupby('embarked').me
        data.loc[data['embarked'].isna(), 'embarked'] = 'S'

        # remove outliers in fare
        data = data[data['fare'] < np.percentile(data['fare'], 97.5)]


        # test data
        test = test_data.drop(['cabin', 'ticket'], axis=1)
        test['age'] = test[['age']].fillna(data['age'].mean())
```

```
In [6]: data.boxplot();
```




```
In [7]: dat = np.array(pd.get_dummies(data.copy()))
        train, valid = split(dat, 100)

        X_train_titanic, y_train_titanic = train[:, 1:], train[:, 0]
        X_val_titanic, y_val_titanic = valid[:, 1:], valid[:, 0]
```

```
In [8]: tree_titanic = DecisionTree(max_depth=2, threshold=.015)
        tree_titanic.fit(X_train_titanic, y_train_titanic)
```

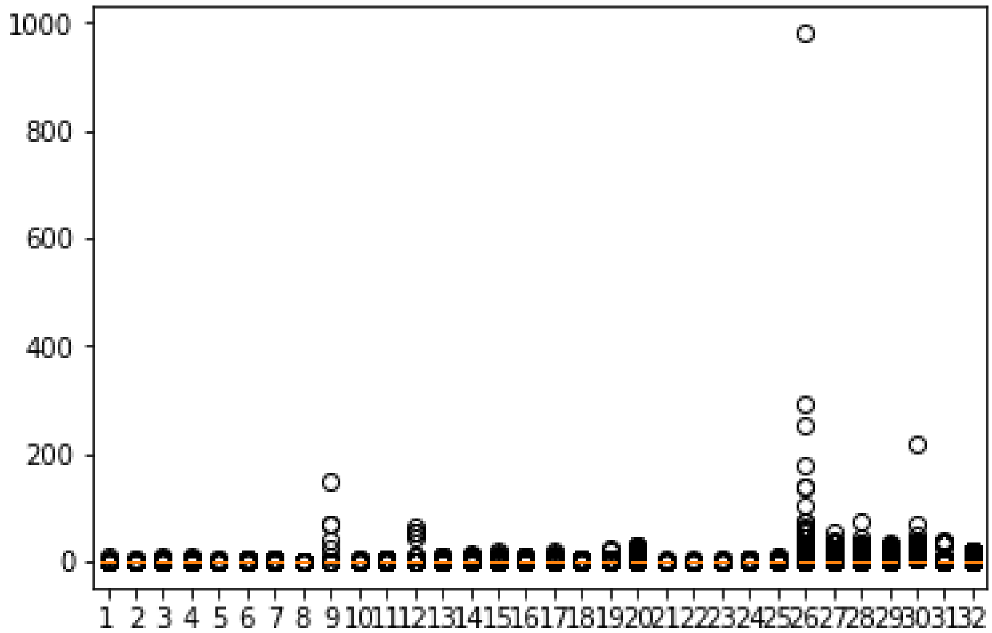
```
In [9]: rf_titanic = RandomForest(max_depth=1000, n_estimators=100)
        rf_titanic.fit(X_train_titanic, y_train_titanic)
```

```
In [10]: X_test_titanic = np.array(pd.get_dummies(test))

         pred = rf_titanic.predict(X_test_titanic)
         pred = pred.astype(int)

         sub = pd.DataFrame(pred, columns=['Category'], index=np.arange(1, len(pred)+1, 1), dty
         sub.index.name = 'Id'

         sub.to_csv('./submission.csv')
```

### 0.0.2 Spam-Ham

```
In [11]: dataset = "spam"

         if dataset == "titanic":
             # Load titanic data
             path_train = 'datasets/titanic/titanic_training.csv'
             data = pd.read_csv(path_train)
             path_test = 'datasets/titanic/titanic_testing_data.csv'
             test_data = pd.read_csv(path_test)
             y = data['survived']
             class_names = ["Died", "Survived"]

         elif dataset == "spam":
             features = [
                 "pain", "private", "bank", "money", "drug", "spam", "prescription",
                 "creative", "height", "featured", "differ", "width", "other",
                 "energy", "business", "message", "volumes", "revision", "path",
                 "meter", "memo", "planning", "pleased", "record", "out",
                 "semicolon", "dollar", "sharp", "exclamation", "parenthesis",
                 "square_bracket", "ampersand"
             ]
             assert len(features) == 32

             # Load spam data
             path_train = 'datasets/spam-dataset/spam_data.mat'
             data = scipy.io.loadmat(path_train)
             X = data['training_data']
             y = np.squeeze(data['training_labels'])
             Z = data['test_data']
             class_names = ["Ham", "Spam"]

         else:
             raise NotImplementedError("Dataset %s not handled" % dataset)
```

```
In [12]: plt.boxplot(X);
```




```
In [13]: # outlier values for all features
         idx = np.all(X <= np.percentile(X, 97.5, axis=0), axis=1)

         X = X[idx]
         y = y[idx]

In [14]: data = np.hstack((X, y.reshape(-1, 1)))
         train, valid = split(data, .2)

         X_train_spam, y_train_spam = train[:, :-2], train[:, -1]
         X_valid_spam, y_valid_spam = valid[:, :-2], valid[:, -1]
         X_test_spam = Z

In [47]: tree_spam = DecisionTree(max_depth=4, threshold=.02)
         tree_spam.fit(X_train_spam, y_train_spam)

In [16]: rf_spam = RandomForest(max_depth=1000, n_estimators=300, n_random_features=10)
         rf_spam.fit(X_train_spam, y_train_spam)

In [50]: pred = rf_spam.predict(X_test_spam)
         pred = pred.astype(int)

         sub = pd.DataFrame(pred, columns=['Category'], index=np.arange(1, len(pred)+1, 1), dty
         sub.index.name = 'Id'

         sub.to_csv('./submission.csv')
```

### 0.1 2.3

**1.** I used the mean value for numerical missing values. I removed cabin and ticket as, in my opinion, would not affect much to the outcome. After that there was no nan value in categorical features.

**2.** If improvement (current node's gini impurity - child nodes') was greater than .015, I stopped splitting or else kept splitting until fully grown and prune with same logic.

**3.** Simply used a list of decision trees each with different set of features.

**4.** I don't think so.

**5.** No… I'm out of ideas.. What a fooooool.…

### 0.2 2.4 Performances

```
In [18]: print(f'Decision Tree on Titanic Training : {tree_titanic.accuracy(X_train_titanic, y_
         Decision Tree on Titanic Validation : {tree_titanic.accuracy(X_val_titanic, y_val_tita
```

```
Decision Tree on Titanic Training : 0.7938144329896907
Decision Tree on Titanic Validation : 0.75
```

```
In [19]: print(f'Random Forest on Titanic Training : {rf_titanic.accuracy(X_train_titanic, y_tr
         Random Forest on Titanic Validation : {rf_titanic.accuracy(X_val_titanic, y_val_titan
```

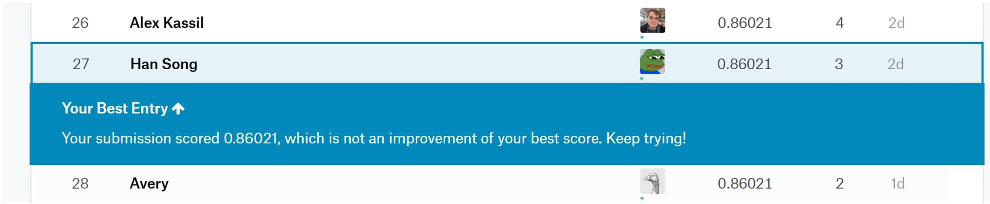
```
Random Forest on Titanic Training : 0.8751431844215349
Random Forest on Titanic Validation : 0.76
```

```
In [20]: print(f'Decision Tree on Spam Training : {tree_spam.accuracy(X_train_spam, y_train_spa
         Decision Tree on Spam Validation : {tree_spam.accuracy(X_valid_spam, y_valid_spam)}')
```

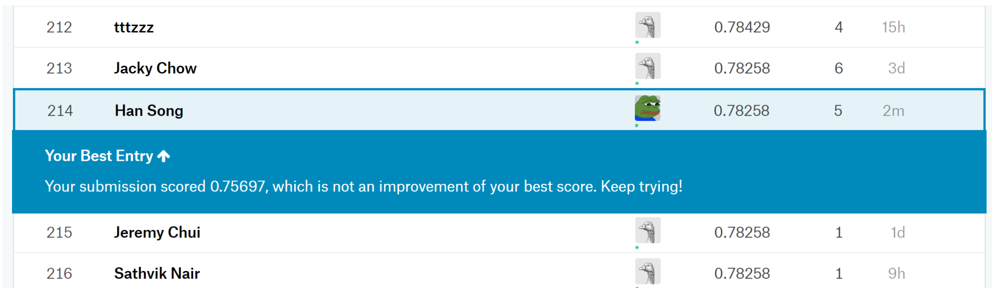
```
Decision Tree on Spam Training : 0.8002658690594882
Decision Tree on Spam Validation : 0.8085106382978723
```

```
In [51]: print(f'Random Forest on Spam Training : {rf_spam.accuracy(X_train_spam, y_train_spam)
         Random Forest on Spam Validation : {rf_spam.accuracy(X_valid_spam, y_valid_spam)}')
```

```
Random Forest on Spam Training : 0.8348288467929544
Random Forest on Spam Validation : 0.8058510638297872
```




titanic.png




spam

## 0.3 2.5

```
In [22]: single_spam = X_train_spam[30]
         single_spam

Out[22]: array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 1.,
                0., 0., 1., 0., 0., 0., 0., 0., 0., 3., 0., 0., 1., 0.])

In [23]: dec = DecisionTree(max_depth=4, threshold=.01)
         dec.fit(X_train_spam, y_train_spam)

In [24]: dec

Out[24]:
         Depth : 0, Split Feature : 28, Split Point : 1.0, Gain : 0.053420098256930226
           Depth : 1, Split Feature : 29, Split Point : 1.0, Gain : 0.013287901979625094
             Depth : 2, Split Feature : 19, Split Point : 1.0, Gain : 0.019579639101447888
               Depth : 3, Label : 0
               Depth : 3, Label : 0
             Depth : 2, Label : 0
           Depth : 1, Split Feature : 19, Split Point : 1.0, Gain : 0.046734117230732675
             Depth : 2, Split Feature : 29, Split Point : 1.0, Gain : 0.020001679772411518
               Depth : 3, Split Feature : 16, Split Point : 1.0, Gain : 0.021617585724248656
                 Depth : 4, Label : 1
                 Depth : 4, Label : 0
               Depth : 3, Split Feature : 3, Split Point : 1.0, Gain : 0.0404505000061805
                 Depth : 4, Label : 0
                 Depth : 4, Label : 1
             Depth : 2, Label : 0

In [25]: np.where(single_spam > 0)
```

```
Out[25]: (array([16, 19, 26, 29], dtype=int64),)
```

```
In [26]: np.array(features)[np.where(single_spam>0)]
```

```
Out[26]: array(['volumes', 'meter', 'dollar', 'parenthesis'], dtype='<U14')
```

```
In [27]: features[28]
```

```
Out[27]: 'exclamation'
```

```
In [28]: dec.predict(single_spam.reshape(1, -1))
```

```
Out[28]: array([0])
```

1. exclamation <= 1 (moved to left node)
2. parenthesis <= 1 (moved to left)
3. parenthesis <= 1 (moved to left)
4. It is not a spam

```
In [29]: y_train_spam[30]
```

```
Out[29]: 0.0
```

## 0.4 2.6

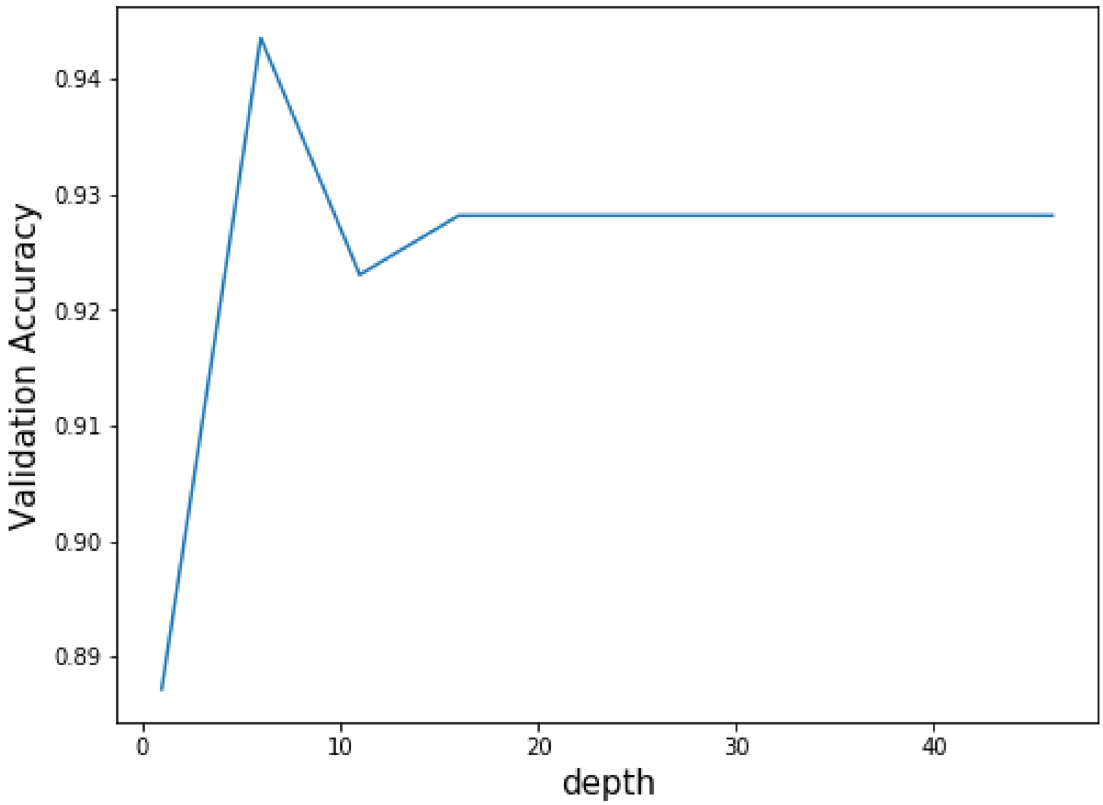
```
In [30]: titanic_train, titanic_valid = split(dat, .2)
```

```
In [31]: titanic_train.shape, titanic_valid.shape
```

```
Out[31]: ((778, 11), (195, 11))
```

```
In [32]: depth = np.arange(1, 50, 5)
         acc = []

         for d in depth:

             dec = DecisionTree(max_depth=d, threshold=.015, prune=False)
             dec.fit(titanic_train[:, :-2], titanic_train[:, -1])

             acc.append(dec.accuracy(titanic_valid[:, :-2], titanic_valid[:, -1]))
```

```
In [33]: plt.figure(figsize=(8,6))
         plt.plot(depth, acc)
         plt.xlabel('depth', size=15)
         plt.ylabel('Validation Accuracy', size=15);
```




Without pruning, it peaks around before the depth of 8~10 and after that, it decreases and again after some depth, it has a constant accuracy score. This is due to overfitting and after some number of depths, it will not increase the accuracy.

```
In [ ]: `
```