In [2]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import math

import warnings
warnings.filterwarnings('ignore')
```

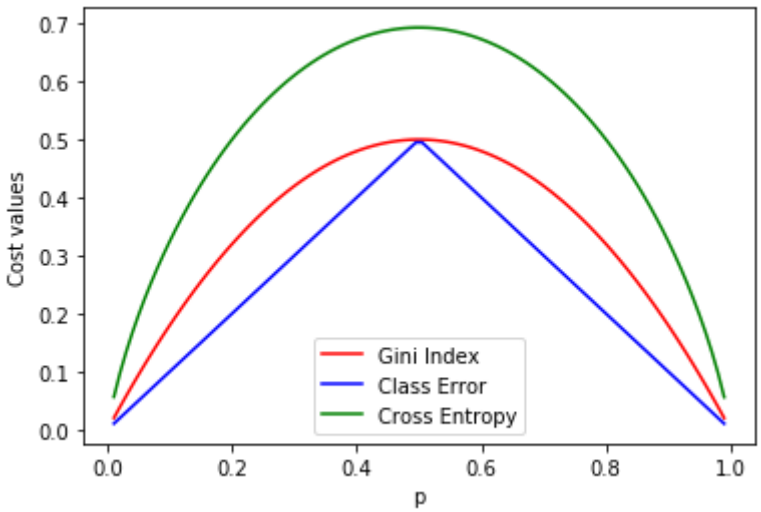
Conceptual: Cost functions for classification trees

1

Consider the Gini index, classification error, and cross-entropy in simple classification settings with two classes. Of these three possible cost functions, which would be best to use when growing a decision tree? Which would be best to use when pruning a decision tree? Why?

In [30]:
```
p = np.linspace(0.01, 1, 100, endpoint=False)
data = np.c_[p, 1-p]
gini = 2 * p * (1 - p)
class_error = 1 - np.max(data, axis=1)
def temp(a):
    return - (a[0] * math.log(a[0]) + a[1] * math.log(a[1]))
cross_entropy = np.apply_along_axis(temp, 1, data)

plt.plot(p, gini, 'r', label='Gini Index')
plt.plot(p, class_error, 'b', label='Class Error')
plt.plot(p, cross_entropy, 'g', label='Cross Entropy')
plt.xlabel('p')
plt.ylabel('Cost values')
plt.legend()
plt.show();
```



When building a classification tree, either the Gini index or the entropy are typically used to evaluate the quality of a particular split, since these two approaches are more sensitive to node purity than is the classification error rate.

When pruning the tree, the classification error rate is preferable if prediction accuracy of the final pruned tree is the goal.

That's because of the greedy algorithm the tree learning is taking, so trying to maximize classification accuracy at each step may not end up selecting the accuracy-maximizing classifier overall. This is exacerbated because classification accuracy is insensitive/noisy (as is depicted above). Hence, to optimize classification accuracy, may lead to fit on noise and overfitting. By contrast, doing accuracy-based pruning at the end is less prone to the fitting-on-noise issue because you're making fewer choices, so the consideration of maximizing your loss function directly is more important.

Application: Predicting attitudes towards racist college professors

learning obejectives:

1. Implement a battery of learners (including trees)
2. Tune hyperparameters
3. Substantively evaluate models

In [3]:
```
train = pd.read_csv('gss_train.csv')
test = pd.read_csv('gss_test.csv')
train.head()
```

Out[3]:

	age	attend	authoritarianism	black	born	childs	colath	colrac	colcom	colmil	...	partyid
0	21	0	4	0	0	0	1	1	0	1	...	
1	42	0	4	0	0	2	0	1	1	0	...	
2	70	1	1	1	0	3	0	1	1	0	...	
3	35	3	2	0	0	2	0	1	0	1	...	
4	24	3	6	0	1	3	1	1	0	0	...	

5 rows × 56 columns

In [4]:
```
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import KFold
kf_10 = KFold(n_splits=10, shuffle=False, random_state=1)

from sklearn.pipeline import Pipeline
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import roc_auc_score
from sklearn.metrics import roc_curve, auc
from sklearn import metrics
```

In [5]:
```
X_train, y_train = train.drop('colrac', axis=1), train['colrac']
X_test, y_test = test.drop('colrac', axis=1), test['colrac']
```

2

Estimate the following models, predicting colrac using the training set (the training .csv) with 10-fold CV.

Tune the relevant hyperparameters for each model as necessary. Only use the tuned model with the best performance for the remaining exercises. Be sure to leave sufficient time for hyperparameter tuning. Grid searches can be computationally taxing and take quite a while, especially for tree-aggregation methods.

a. Logistic Regression

reference: https://github.com/finnqiao/bank-logistic/blob/master/bank-logistic-v2.ipynb (https://github.com/finnqiao/bank-logistic/blob/master/bank-logistic-v2.ipynb)

In [8]:
```
from sklearn.linear_model import LogisticRegression
from sklearn.linear_model import LogisticRegressionCV
```

In [9]:
```
clf_lr = LogisticRegressionCV(cv=10, random_state=0).fit(X_train, y_trai
n)
```

In [44]:
```
evaluate('Logistic Regression', clf_lr)
```

In [11]:
```
# another way to do hyperparameter tuning: GridSearchCV
print(LogisticRegression().get_params().keys())
```

```
dict_keys(['C', 'class_weight', 'dual', 'fit_intercept', 'intercept_sca
ling', 'l1_ratio', 'max_iter', 'multi_class', 'n_jobs', 'penalty', 'ran
dom_state', 'solver', 'tol', 'verbose', 'warm_start'])
```

In [12]:
```
# Create param grid.
para_lr = {'penalty' : ['l1', 'l2'],
           'C' : [100, 10, 1.0, 0.1, 0.01],
           'solver' : ['newton-cg', 'lbfgs', 'liblinear']}

# Create grid search object
lr = GridSearchCV(LogisticRegression(), param_grid=para_lr, cv=10,
                  verbose=1, n_jobs=-1, scoring='accuracy')

# Fit on data
lr.fit(X_train, y_train)
lr.best_estimator_
```

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

[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent work
ers.
[Parallel(n_jobs=-1)]: Done  34 tasks      | elapsed:    2.1s
[Parallel(n_jobs=-1)]: Done 300 out of 300 | elapsed:    3.6s finished
```

Out[12]:
```
LogisticRegression(C=10, class_weight=None, dual=False, fit_intercept=T
rue,
                   intercept_scaling=1, l1_ratio=None, max_iter=100,
                   multi_class='auto', n_jobs=None, penalty='l2',
                   random_state=None, solver='lbfgs', tol=0.0001, verbo
se=0,
                   warm_start=False)
```

In [15]:
```
print(lr.best_score_)
lr.best_estimator_.get_params()
```

```
0.7994760066188638
```

Out[15]:
```
{'C': 10,
 'class_weight': None,
 'dual': False,
 'fit_intercept': True,
 'intercept_scaling': 1,
 'l1_ratio': None,
 'max_iter': 100,
 'multi_class': 'auto',
 'n_jobs': None,
 'penalty': 'l2',
 'random_state': None,
 'solver': 'lbfgs',
 'tol': 0.0001,
 'verbose': 0,
 'warm_start': False}
```

b. Naive Bayes

reference:

https://towardsdatascience.com/machine-learning-part-16-naive-bayes-classifier-in-python-c9d3fa496fa4 (https://towardsdatascience.com/machine-learning-part-16-naive-bayes-classifier-in-python-c9d3fa496fa4)

Doesn't require hyperparameter tuning

In [16]:
```
from sklearn.naive_bayes import GaussianNB
clf_nb = GaussianNB().fit(X_train, y_train)
```

In [45]:
```
evaluate('Naive Bayes', clf_nb)
```

c. Elastic Net Regression

In [23]:
```
from sklearn.linear_model import ElasticNet, ElasticNetCV
cv_values = np.arange(0.1, 1.1, 0.1)

clf_en = ElasticNetCV(l1_ratio=cv_values, cv=10, max_iter=1000, random_s
tate=1917)
clf_en.fit(X_train, y_train)
```

Out[23]:
```
ElasticNetCV(alphas=None, copy_X=True, cv=10, eps=0.001, fit_intercept=
True,
             l1_ratio=array([0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.
9, 1. ]),
             max_iter=1000, n_alphas=100, n_jobs=None, normalize=False,
             positive=False, precompute='auto', random_state=1917,
             selection='cyclic', tol=0.0001, verbose=0)
```

In [20]:
```
print(str(clf_en.alpha_))
print(str(clf_en.l1_ratio_))
```

```
0.008965265581870592
0.2
```

In [98]:
```
model_data['name'].append('Elastic Net')
fpr, tpr, thresholds = roc_curve(y_train, clf_en.predict(X_train))
Roc_auc = auc(fpr,tpr)
model_data['ROC_auc'].append(Roc_auc)
cv_er = cross_val_score(clf_en, X_train, y_train, cv=kf_10,
                        scoring='r2').mean()
model_data['cv_er'].append(cv_er)
```

d. Decision Tree (CART)

pruning reference:

https://scikitlearn.org/stable/auto_examples/tree/plot_cost_complexity_pruning.html (https://scikitlearn.org/stable/auto_examples/tree/plot_cost_complexity_pruning.html)

In [31]:
```
from sklearn.tree import DecisionTreeClassifier
print(DecisionTreeClassifier().get_params().keys())
```

```
dict_keys(['ccp_alpha', 'class_weight', 'criterion', 'max_depth', 'max_
features', 'max_leaf_nodes', 'min_impurity_decrease', 'min_impurity_spl
it', 'min_samples_leaf', 'min_samples_split', 'min_weight_fraction_lea
f', 'presort', 'random_state', 'splitter'])
```

In [32]:
```
para_dt = {'max_depth': np.arange(1, 56),
           'min_samples_leaf': np.arange(1, 11)}
clf_dt = GridSearchCV(DecisionTreeClassifier(), para_dt,
                      scoring='accuracy', cv=10)
clf_dt.fit(X_train, y_train)
clf_dt.best_estimator_
```

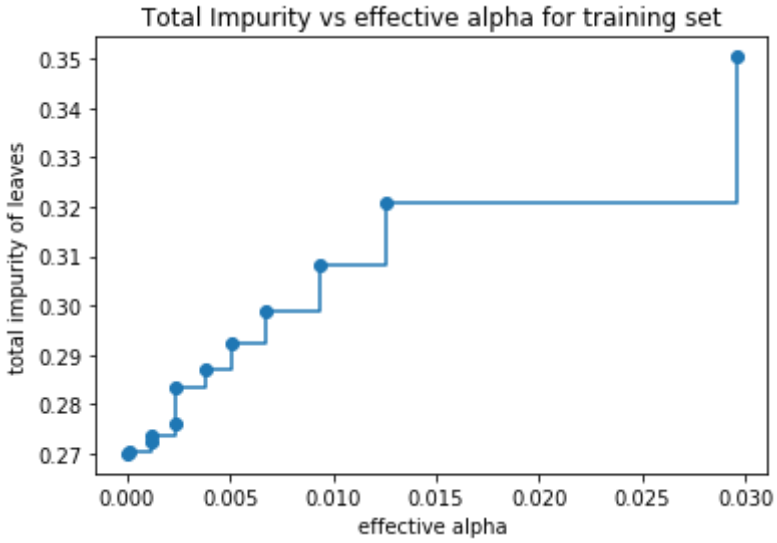
Out[32]:
```
DecisionTreeClassifier(ccp_alpha=0.0, class_weight=None, criterion='gin
i',
                       max_depth=4, max_features=None, max_leaf_nodes=N
one,
                       min_impurity_decrease=0.0, min_impurity_split=No
ne,
                       min_samples_leaf=5, min_samples_split=2,
                       min_weight_fraction_leaf=0.0, presort='deprecate
d',
                       random_state=None, splitter='best')
```

In [47]:
```
evaluate('unpruned Decision Tree', clf_dt.best_estimator_)
```

In [49]:
```
# pruning
path = clf_dt.best_estimator_.cost_complexity_pruning_path(X_train, y_tr
ain)
ccp_alphas, impurities = path.ccp_alphas, path.impurities

fig, ax = plt.subplots()
ax.plot(ccp_alphas[:-1], impurities[:-1], marker='o', drawstyle="steps-p
ost")
ax.set_xlabel("effective alpha")
ax.set_ylabel("total impurity of leaves")
ax.set_title("Total Impurity vs effective alpha for training set")
```

Out[49]:
```
Text(0.5, 1.0, 'Total Impurity vs effective alpha for training set')
```

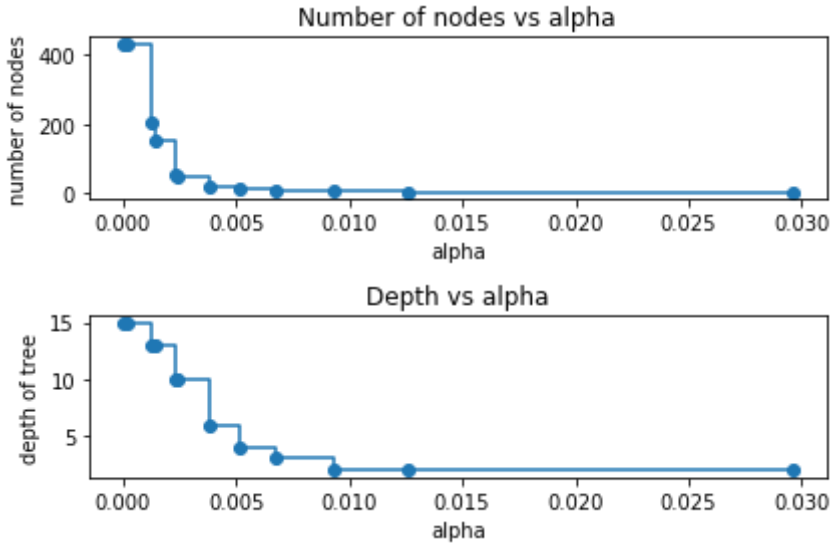


In [19]:
```
clfs = []
for ccp_alpha in ccp_alphas:
    clf = DecisionTreeClassifier(random_state=0, ccp_alpha=ccp_alpha)
    clf.fit(X_train, y_train)
    clfs.append(clf)
print("Number of nodes in the last tree is: {} with ccp_alpha: {}".forma
t(
      clfs[-1].tree_.node_count, ccp_alphas[-1]))
```

```
Number of nodes in the last tree is: 1 with ccp_alpha: 0.14835462817211
664
```

In [20]:
```
clfs = clfs[:-1]
ccp_alphas = ccp_alphas[:-1]

node_counts = [clf.tree_.node_count for clf in clfs]
depth = [clf.tree_.max_depth for clf in clfs]
fig, ax = plt.subplots(2, 1)
ax[0].plot(ccp_alphas, node_counts, marker='o', drawstyle="steps-post")
ax[0].set_xlabel("alpha")
ax[0].set_ylabel("number of nodes")
ax[0].set_title("Number of nodes vs alpha")
ax[1].plot(ccp_alphas, depth, marker='o', drawstyle="steps-post")
ax[1].set_xlabel("alpha")
ax[1].set_ylabel("depth of tree")
ax[1].set_title("Depth vs alpha")
fig.tight_layout()
```



In [ ]:
```
```

e. Bagging

In [50]:
```
from sklearn.ensemble import BaggingClassifier, RandomForestClassifier
X_train.shape
```

Out[50]:
```
(1476, 55)
```

In [51]:
```
print(BaggingClassifier().get_params().keys())
```

```
dict_keys(['base_estimator', 'bootstrap', 'bootstrap_features', 'max_features', 'max_samples', 'n_estimators', 'n_jobs', 'oob_score', 'random_state', 'verbose', 'warm_start'])
```

In [55]:
```
para_bag = {'n_estimators': [10, 100, 1000]}
clf_bag = GridSearchCV(BaggingClassifier(max_samples=0.6), para_bag,
                       scoring='accuracy', cv=10, n_jobs=-1)
clf_bag.fit(X_train, y_train)
clf_bag.best_estimator_
```

Out[55]:
```
BaggingClassifier(base_estimator=None, bootstrap=True, bootstrap_featur
es=False,
                  max_features=1.0, max_samples=0.6, n_estimators=1000,
                  n_jobs=None, oob_score=False, random_state=None, verb
ose=0,
                  warm_start=False)
```

In [56]:
```
evaluate('Bagging ', clf_bag.best_estimator_)
```

f. Random Forest

In [75]:
```
import math
math.sqrt(55)
```

Out[75]:
```
7.416198487095663
```

In [90]:
```
from sklearn.model_selection import RepeatedStratifiedKFold
para_rf = {'n_estimators': [10, 100, 1000],
           'max_features': np.arange(1, 8)}
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
clf_rf = GridSearchCV(RandomForestClassifier(), para_rf,
                      scoring='accuracy', cv=cv, n_jobs=-1)
clf_rf.fit(X_train, y_train)
clf_rf.best_estimator_
```

Out[90]:
```
RandomForestClassifier(bootstrap=True, ccp_alpha=0.0, class_weight=Non
e,
                       criterion='gini', max_depth=None, max_features=
7,
                       max_leaf_nodes=None, max_samples=None,
                       min_impurity_decrease=0.0, min_impurity_split=No
ne,
                       min_samples_leaf=1, min_samples_split=2,
                       min_weight_fraction_leaf=0.0, n_estimators=1000,
                       n_jobs=None, oob_score=False, random_state=None,
                       verbose=0, warm_start=False)
```

In [91]:
```
evaluate('random forest', clf_rf.best_estimator_)
```

g. Boosting

In [60]:
```
from sklearn.ensemble import GradientBoostingClassifier
```

In [61]:
```
para_bst = {'n_estimators': [10, 100, 1000],
            'learning_rate': [0.001, 0.01, 0.1],
            'subsample': [0.5, 0.7, 1.0],
            'max_depth': [3, 7, 9]}
clf_bst = GridSearchCV(GradientBoostingClassifier(), para_bst,
                       scoring='accuracy', cv=10, n_jobs=-1)
clf_bst.fit(X_train, y_train)
clf_bst.best_estimator_
```

Out[61]:
```
GradientBoostingClassifier(ccp_alpha=0.0, criterion='friedman_mse', ini
t=None,
                           learning_rate=0.01, loss='deviance', max_dep
th=3,
                           max_features=None, max_leaf_nodes=None,
                           min_impurity_decrease=0.0, min_impurity_spli
t=None,
                           min_samples_leaf=1, min_samples_split=2,
                           min_weight_fraction_leaf=0.0, n_estimators=1
000,
                           n_iter_no_change=None, presort='deprecated',
                           random_state=None, subsample=0.7, tol=0.000
1,
                           validation_fraction=0.1, verbose=0,
                           warm_start=False)
```

In [62]:
```
evaluate('Boosting', clf_bst.best_estimator_)
```

Evaluate the models

3

Compare and present each model's (training) performance based on

```
Cross-validated error rate
ROC/AUC
```

In [42]:
```
model_data = {'name': [],
              'cv_er':[],
              'ROC_auc':[]}
```

In [43]:
```
def evaluate(name, clf):
    model_data['name'].append(name)
    fpr, tpr, thresholds = roc_curve(y_train, clf.predict(X_train))
    Roc_auc = auc(fpr,tpr)
    model_data['ROC_auc'].append(Roc_auc)
    cv_er = cross_val_score(clf, X_train, y_train, cv=kf_10,
                            scoring='accuracy').mean()
    model_data['cv_er'].append(1 - cv_er)
```

In [101]:
```
evaluation.sort_values(by='ROC_auc', axis=0, ascending=False)
```

Out[101]:

	name	cv_er	ROC_auc
10	random forest	0.201255	1.000000
5	Bagging	0.205327	0.992358
7	Boosting	0.193119	0.896279
11	Elastic Net	0.404517	0.891444
0	Logistic Regression	0.202588	0.816748
3	unpruned Decision Tree	0.221626	0.790820
1	Naive Bayes	0.264902	0.743125

4

Which is the best model? Defend your choice.

For classification models, AUC is a better measure of classifier performance than accuracy because it does not bias on size of test or evaluation data. Consequently, Random Forest is the best classifier for this data set.

5

Evaluate the final, best model's (selected in 4) performance on the test set (the test .csv) by calculating and presenting the classification error rate and AUC. Compared to the fit evaluated on the training set in questions 3-4, does the "best" model generalize well? Why or why not? How do you know?

In [95]:
```
fpr, tpr, thresholds = roc_curve(y_test, clf_rf.best_estimator_.predict(
X_test))
Roc_auc_rf = auc(fpr,tpr)
print(Roc_auc_rf)
```

```
0.7930486593843098
```

In [96]:
```
cv_er_rf = cross_val_score(clf_rf.best_estimator_, X_test, y_test,
                           cv=kf_10, scoring='accuracy').mean()
print(cv_er_rf)
```

```
0.7991836734693878
```

In [97]:
```
1 - cv_er_rf
```

Out[97]:
```
0.2008163265306122
```

Despite the classification error rate does not change much, the AUC calculated by the test data drops a lot from the one obtained based on the training data. One possible explanation is that, Random forests have many many degrees of freedom, so it is relatively easy for them to get to the point that they have near 100% accuracy in-sample. That is to say, overfitting. I tried Gridsearch and repeated sampling with RepeatedStratifiedKFold, yet the training AUC remains high.

Hence, removing redundant features by analysing the models feature importance scores may be necessary.

In [ ]: