Rim_Nak Won_HW5

March 1, 2020

Problem Set 5

MACS 30100 Winter 2020

Nak Won Rim

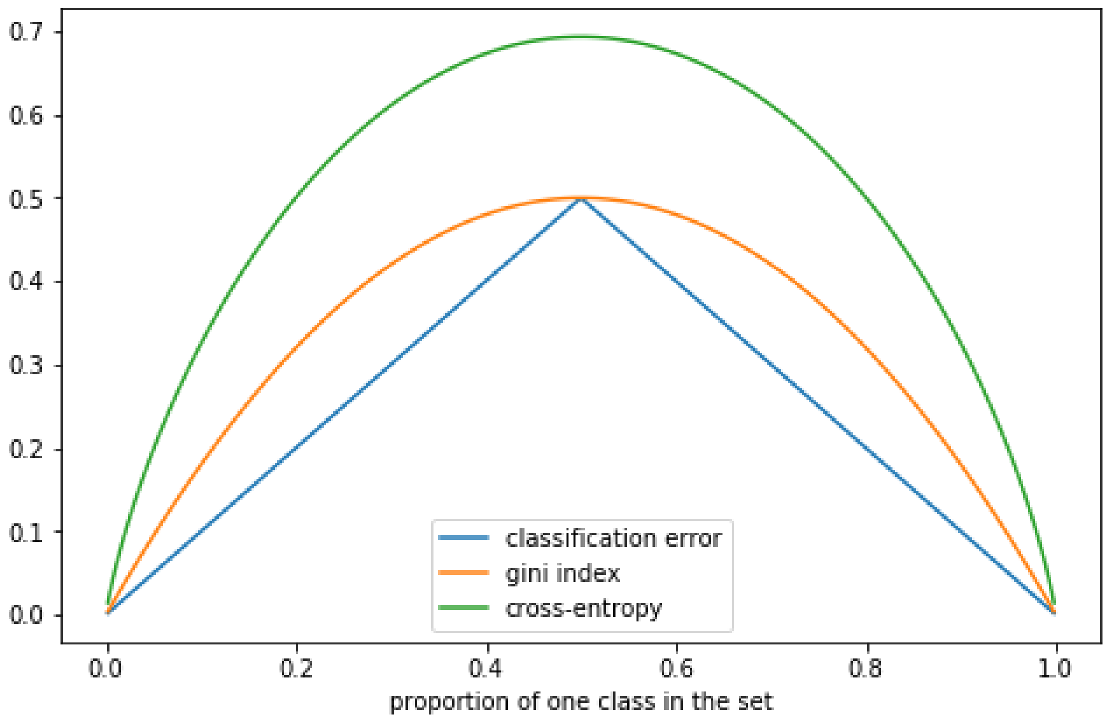
[1]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import GridSearchCV, cross_val_score
from sklearn.linear_model import LogisticRegression, ElasticNetCV, SGDClassifier
from sklearn.naive_bayes import GaussianNB
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import BaggingClassifier, RandomForestClassifier,␣
 ↪GradientBoostingClassifier
from sklearn.metrics import roc_auc_score, roc_curve
from sklearn.inspection import plot_partial_dependence
```

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?

The classification error rate is a little bit different from the Gini index or the cross-entropy since it is more related to the accuracy of the prediction while the other two are more related to the impurity of the node. In addition, cross-entropy is more sensitive to the impurity of the node than the Gini index as the following plot shows:

[2]:
```
p_1 = np.linspace(0,1,500)[1:-1]
p_2 = 1 - p_1
classification_error = np.array([1 - max(p, 1 - p) for p in p_1])
gini_index = (p_1 * p_2) * 2
cross_entropy = - ((p_1 * np.log(p_1)) + (p_2 * np.log(p_2)))
plt.figure(figsize=(8,5))
sns.lineplot(p_1, classification_error)
```

```python
sns.lineplot(p_1, gini_index)
sns.lineplot(p_1, cross_entropy)
plt.legend(['classification error', 'gini index', 'cross-entropy'])
plt.xlabel('proportion of one class in the set');
```



The primary goal when growing a decision tree is to build the biggest tree in a reasonable range. Therefore, I think using a method that is most sensitive to node impurity will be a good choice. In other words, I think cross-entropy, which is the measure most robust to node impurity, will be the best choice in growing a decision tree. In addition, as we discussed in class on Monday (slide 18), methods using nodes impurity tend to grow more accurate trees than using the classification error rate. This could be related to that we are using a greedy approach when growing a tree – measuring impurity is a little less vulnerable to choosing a split that makes sense in the current node but not in the bigger picture.

When pruning the tree, the primary goal is to find an optimal tree with a small number of nodes (for interpretability and preventing overfitting) and high accuracy. In this case, pruning based on the classification error rate could be more effective. Since it is less robust to node impurity, pruning the tree based on classification error will remove the splits that are beneficial to getting more pure nodes but not beneficial to accuracy. A good example of this is that when using impurity measures to grow a tree, a tree could have a split that produces two nodes that share the same prediction (as we saw on Monday's R code). Using the classification error rate in the pruning process will remove such nodes. Also, the end goal of decision tree is often producing accurate prediction, so using a measure that is directly related to the accuracy in pruning will help achieving this goal.

To sum up, when growing the decision tree, it is best to use a measure that is more sensitive to node impurity, so we can grow a bigger tree. This makes the cross-entropy the best method. On the contrary, when pruning the tree, we want to make the tree as succinct and accurate as possible. The classification error rate is a good choice here since is relatively unaffected by node impurity and directly measures the accuracy of the tree.

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

Note that the CV-estimated accuracy rates are also calculated in each model except elastic net regression (where the SKLearn uses MSE), but I am only going to report the hyperparameters (if any) for the models in this section. SKLearn has a lot of hyperparameters for each model, but I am mainly going to focus on hyperparameters we discussed in class since grid search for hyperparameters are, as the assignment instruction noted, quite computationally expensive. For example, SKLearn has C hyperparameter for the logistic regression model, but I did not optimize it because we did not discuss it in class.

```
[3]: # loading data
     gss_train = pd.read_csv('data/gss_train.csv')
     gss_test = pd.read_csv('data/gss_test.csv')
     train_X = gss_train.drop('colrac', axis=1)
     train_Y = gss_train['colrac']
     test_X = gss_test.drop('colrac', axis=1)
     test_Y = gss_test['colrac']
```

2.1 Logistic regression

```
[4]: lr_gridcv = GridSearchCV(estimator=LogisticRegression(solver='liblinear'),
                              param_grid={}, cv=10, refit=True, n_jobs=4).
      ↪fit(train_X, train_Y)
```

2.2 Naive Bayes

```
[5]: nb_gridcv = GridSearchCV(estimator=GaussianNB(), param_grid={}, cv=10,
                              refit=True, n_jobs=-1).fit(train_X, train_Y)
```

2.3 Elastic net regression

Just to clarify in case there is any confusion (especially for future me), the alpha value in SKLearn elastic net regression is equivalent to lambda in glmnet in R, and l1 ratio in SKLearn elastic net regression is equivalent to alpha in glmnet in R.

```
[6]: elnet_gridcv = ElasticNetCV(cv=10).fit(train_X, train_Y)
     print("best alpha by CV is:", elnet_gridcv.alpha_)
     print("best l1 ratio by CV is:", elnet_gridcv.l1_ratio_)
```

```
best alpha by CV is: 0.0038452641680228584
best l1 ratio by CV is: 0.5
```

2.4 Decision tree (CART)

The minimum ratio of samples required to be at a terminal node is the ratio, not the number of samples required for each terminal node. For example, if the ratio is 0.1 and the sample is 100, it means that at least 10 samples should be contained in each terminal node.

[7]:
```
cart_grid = {'criterion': ['gini', 'entropy'],
             'max_depth': range(2, 20),
             'min_samples_leaf': np.linspace(0.1, 0.5, 5)}
cart_gridcv = GridSearchCV(estimator=DecisionTreeClassifier(),␣
 ↪param_grid=cart_grid,
                           cv=10, refit=True, n_jobs=4).fit(train_X, train_Y)
print("best split criterion:", cart_gridcv.best_estimator_.criterion)
print("best maximum depth:", cart_gridcv.best_estimator_.max_depth)
print("best minimum ratio of samples required to be at a terminal node:",␣
 ↪cart_gridcv.best_estimator_.min_samples_leaf)
```

```
best split criterion: entropy
best maximum depth: 3
best minimum ratio of samples required to be at a terminal node: 0.2
```

2.5 Bagging

[8]:
```
bagg_grid = {'n_estimators': range(10,100)}
bagg_gridcv = GridSearchCV(estimator=BaggingClassifier(), param_grid=bagg_grid,
                           cv=10, refit=True, n_jobs=4).fit(train_X, train_Y)
print('best number of base estimators in the ensemble (number of bootstrap␣
 ↪sample; B):', bagg_gridcv.best_estimator_.n_estimators)
```

```
best number of base estimators in the ensemble (number of bootstrap sample; B):
86
```

2.6 Random forest

Similar to what I mentioned in CART, the minimum ratio of features to consider when looking for the best split is a ratio, not a number. For example, if the ratio is 0.1 and the number of features is 50, 5 features will be considered in each split. The same logic is applied to minimum ratio of samples required to be at a terminal node (also analogous to CART)

[9]:
```
rf_grid = {'n_estimators': range(50, 501, 50),
           'max_features': np.linspace(0.1, 0.5, 5),
           'min_samples_leaf': np.linspace(0.1, 0.5, 5),
           'max_leaf_nodes': range(50, 501, 50)}
rf_gridcv = GridSearchCV(estimator=RandomForestClassifier(), param_grid=rf_grid,
                         cv=10, refit=True, n_jobs=4).fit(train_X, train_Y)
print("best number of trees in the forest:", rf_gridcv.best_estimator_.
 ↪n_estimators)
print("best ratio of features to consider when looking for the best split:",␣
 ↪rf_gridcv.best_estimator_.max_features)
```

```
print("best minimum ratio of samples required to be at a terminal node:",␣
↪rf_gridcv.best_estimator_.min_samples_leaf)
print("best number of maximum terminal node:", rf_gridcv.best_estimator_.
↪max_leaf_nodes)
```

```
best number of trees in the forest: 50
best ratio of features to consider when looking for the best split:
0.30000000000000004
best minimum ratio of samples required to be at a terminal node: 0.1
best number of maximum terminal node: 450
```

2.7 Boosting

The hyperparameter maximum depth of the tree is not equivalent to the hyperparameter number of splits in each tree (d) we discussed in class, but it does a similar regularization on tree size (when maximum depth = 1, it is equivalent to d = 1)

```
[10]: bst_grid = {'n_estimators': range(50, 501, 50), 'learning_rate': [0.001, 0.005,␣
↪0.01, 0.05, 0.1],
             'max_depth': range(1,3)}
bst_gridcv = GridSearchCV(estimator=GradientBoostingClassifier(),␣
↪param_grid=bst_grid,
                          cv=10, refit=True, n_jobs=4).fit(train_X, train_Y)
print("best number of trees (B):", bst_gridcv.best_estimator_.n_estimators)
print("best learning rate (lambda):", bst_gridcv.best_estimator_.learning_rate)
print("best maximum depth of the tree:", bst_gridcv.best_estimator_.max_depth)
```

```
best number of trees (B): 500
best learning rate (lambda): 0.1
best maximum depth of the tree: 1
```

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

3.1 Cross-validated error rate

Note that SKLearn does not have an option to use the elastic net regression as a classifier (like option family='binomial' in glmnet in R), so I am using a linear model classifier based on the alpha and l1 ratio from the result of CV, and calculating 10-fold CV error rate.

The cross-validated error rate of each model are:

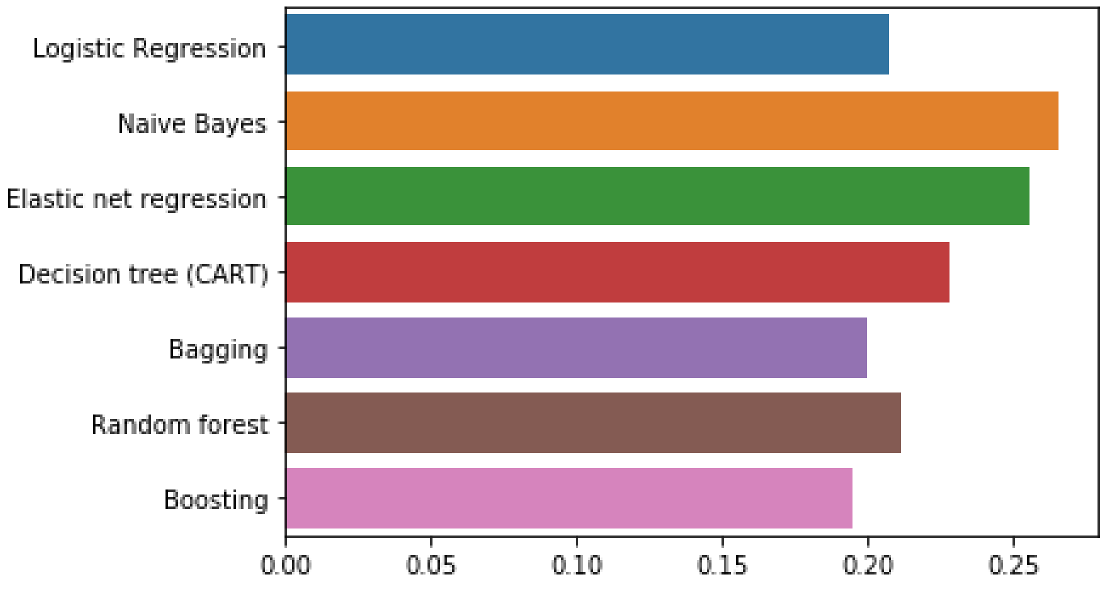
```
[11]: cv_err ={}
cv_err['Logistic Regression'] = 1 - lr_gridcv.best_score_
cv_err['Naive Bayes'] = 1 - nb_gridcv.best_score_
elnet_class = SGDClassifier(alpha = elnet_gridcv.alpha_, l1_ratio =␣
↪elnet_gridcv.l1_ratio_).fit(train_X, train_Y)
```

```
cv_err['Elastic net regression'] = 1 - cross_val_score(elnet_class, train_X,␣
↪train_Y, cv=10).mean()
cv_err['Decision tree (CART)'] = 1 - cart_gridcv.best_score_
cv_err['Bagging'] = 1 - bagg_gridcv.best_score_
cv_err['Random forest'] = 1 - rf_gridcv.best_score_
cv_err['Boosting'] = 1 - bst_gridcv.best_score_
cv_err
```

```
[11]: {'Logistic Regression': 0.20731707317073167,
       'Naive Bayes': 0.26558265582655827,
       'Elastic net regression': 0.2554857041673917,
       'Decision tree (CART)': 0.22831978319783197,
       'Bagging': 0.19986449864498645,
       'Random forest': 0.21138211382113825,
       'Boosting': 0.19444444444444442}
```

```
[12]: sns.barplot(x=list(cv_err.values()), y=list(cv_err.keys()))
```

```
[12]: <matplotlib.axes._subplots.AxesSubplot at 0x2303da40b08>
```



```
[13]: print('the null model accuraccy is:', 1 - sum(train_Y) / len(train_Y))
```

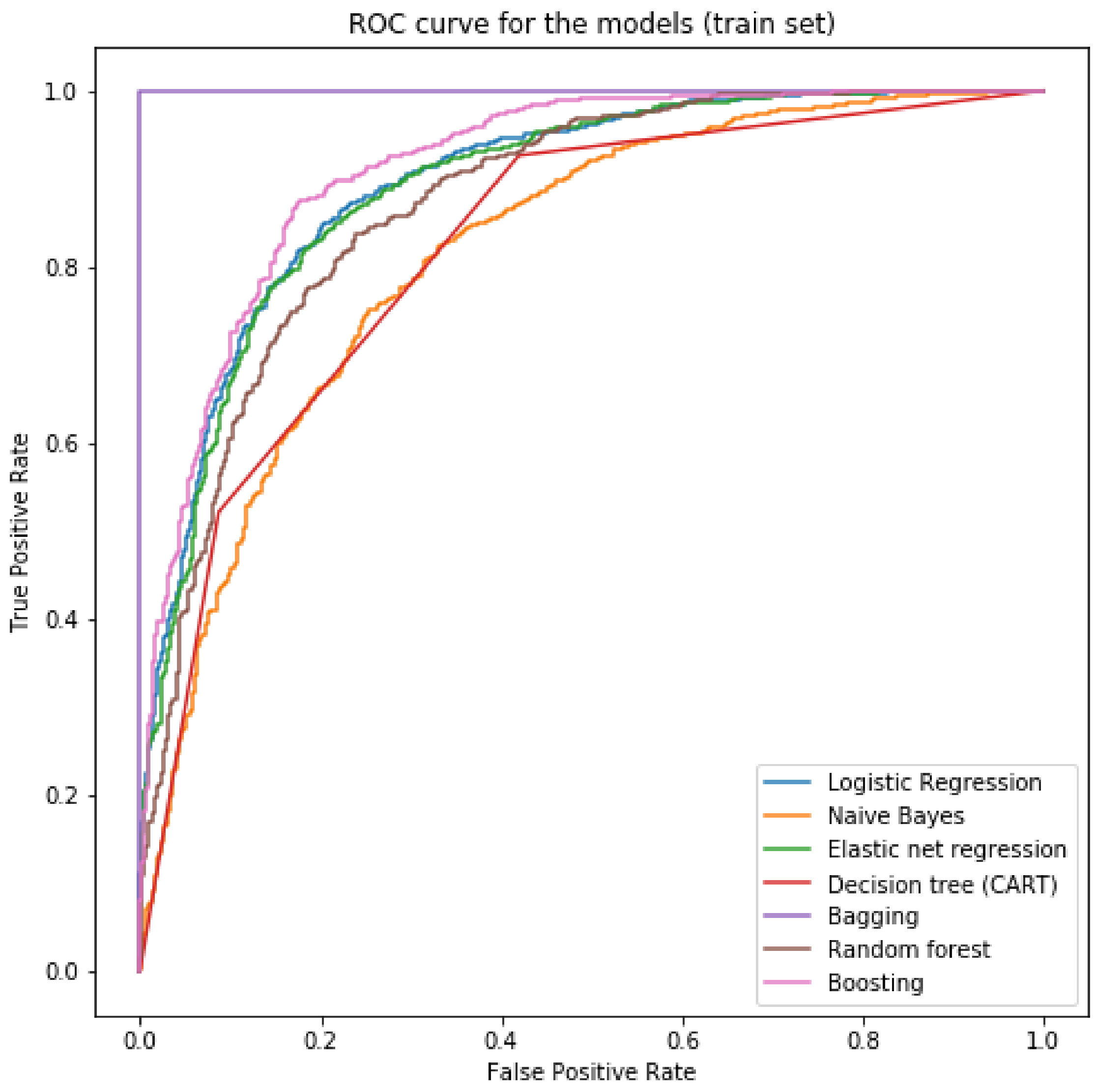
```
the null model accuraccy is: 0.4749322493224932
```

We can see that Boosting has the lowest cross-validated error rate with 0.19. Logistic Regression, Bagging and Random Forest also shows good cross-validated error rate around 0.2. I think this is decent accuracy overall, especially considering the error rate of the null model (predicting everything to be 1) is about 0.48.

3.2 ROC/AUC

The roc curves using the training set for each model are:

[14]:
```
models = {'Logistic Regression': lr_gridcv,
          'Naive Bayes': nb_gridcv,
          'Elastic net regression': elnet_gridcv,
          'Decision tree (CART)': cart_gridcv,
          'Bagging': bagg_gridcv,
          'Random forest': rf_gridcv,
          'Boosting': bst_gridcv}
auc_train = {}
plt.figure(figsize=(8,8))
for name, mod in models.items():
    if name == 'Elastic net regression':
        pred = mod.predict(train_X)
    else:
        pred = mod.predict_proba(train_X)[:, 1]
    auc_train[name] = roc_auc_score(train_Y, pred)
    fp, tp, th = roc_curve(train_Y, pred)
    plt.plot(fp, tp, label=name)
plt.legend()
plt.title('ROC curve for the models (train set)')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate');
```



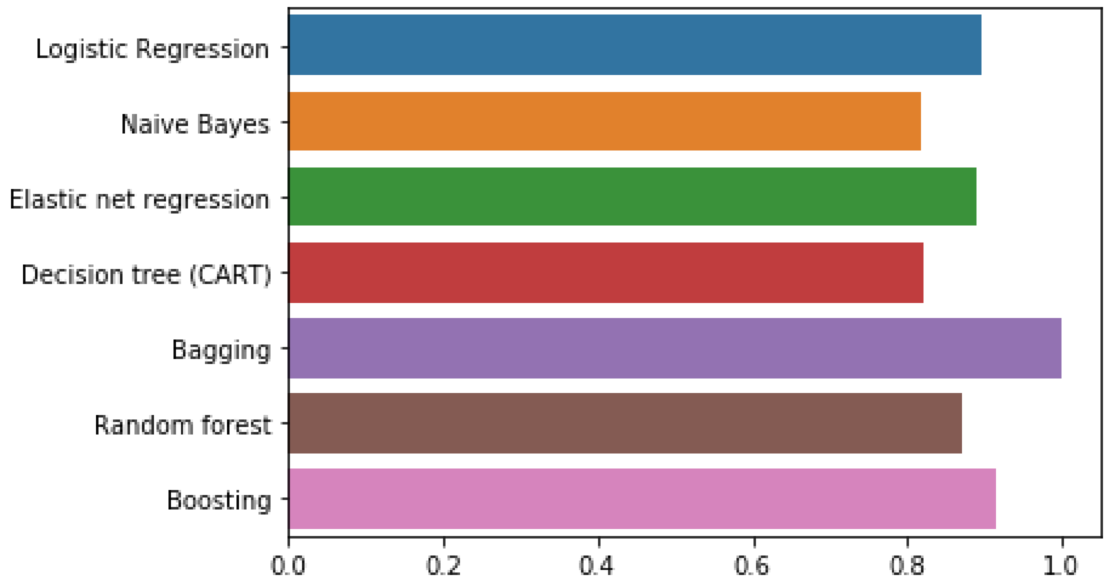
The AUC values calculated on the above ROC curves for each model are:

[15]:
```
auc_train
```

[15]:
```
{'Logistic Regression': 0.8958943444848373,
 'Naive Bayes': 0.816411577930146,
 'Elastic net regression': 0.8914233123188071,
 'Decision tree (CART)': 0.8225309465740187,
 'Bagging': 1.0,
 'Random forest': 0.8727292807510009,
 'Boosting': 0.9138244903593945}
```

[16]:
```
sns.barplot(x=list(auc_train.values()), y=list(auc_train.keys()))
```

```
[16]: <matplotlib.axes._subplots.AxesSubplot at 0x2303e12f4c8>
```



We can see that Bagging actually shows a perfect ROC curve and AUC of 1 in the training set. Boosting, Logistic Regression and Elastic Net Regression seem to be doing a good job as well, having AUC around 0.9. Analogous to the cross-validated error rate, I think these models are showing decent performance in the classification problem.

4 Which is the best model? Defend your choice.

In terms of cross-validated error rate, Boosting performed the best while Logistic Regression, Bagging and Random Forest also performed quite well. In terms of ROC/AUC in training set, Bagging performed extraordinarily well, showing perfect score, while Boosting, Logistic Regression, and Elastic Net Regression also showed good performance. The intersection of the two sets of models that performed well consists of Boosting, Logistic Regression, and Bagging.

As we discussed briefly in week 3, there is no absolute criterion for choosing the best model. Therefore, the best model among these three will depend on what we want from the models, rather than having one model that is better than all others. For example, if we want a method that is computationally inexpensive, Logistic Regression could be a good choice because it is very easy to calculate.

However, if I have to choose, I would say that Boosting (Gradient Boosting) is the best model based on the data so far. Boosting has the best cross-validated error rate, and has second-best ROC/AUC on the training set. It is true that Bagging shows a phenomenally good ROC/AUC on the training set, I think cross-validated measure might be better in evaluating a model because of overfitting. I also think that the AUC of 1 is highly unrealistic and is almost certain that Bagging is overfitting the training data right now. All in all, we will need to examine the result from the test set to draw a final conclusion.

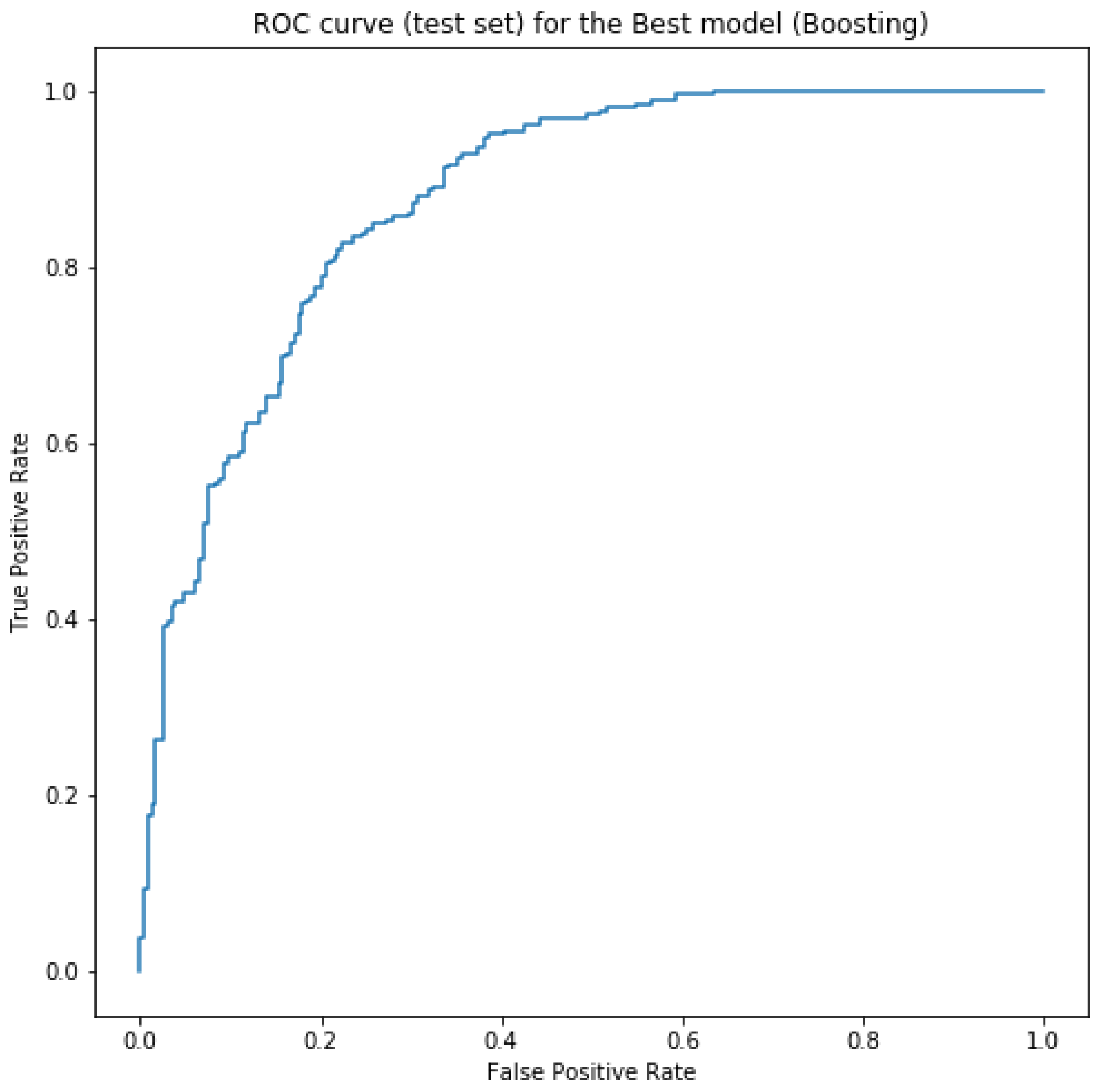
5 Evaluate the *final*, best model's (selected in 4) performance on the test set (the test.csv) by calculating and presenting the classification error rate and AUC. Compared to the fit evaluated on the training set in questions 3-4, does the "best" model generalize well? Why or why not? How do you know?

[17]:
```
test_err_boosting = 1 - sum(bst_gridcv.predict(test_X) == test_Y) / len(test_Y)
print('Test error rate was:', test_err_boosting)
```

```
Test error rate was: 0.21095334685598377
```

[18]:
```
pred = bst_gridcv.predict_proba(test_X)[:, 1]
fp, tp, th = roc_curve(test_Y, pred)
plt.figure(figsize=(8,8))
plt.plot(fp, tp, label='Boosting')
plt.title('ROC curve (test set) for the Best model (Boosting)')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate');
print('AUC in test set for the best model (Boosting):', roc_auc_score(test_Y,␣
 ↪pred))
```

```
AUC in test set for the best model (Boosting): 0.8765971532605098
```



[19]:
```
print('For Boosting:')
print('Cross-validated error rate:', cv_err['Boosting'])
print('Test error rate:', test_err_boosting)
print('AUC for training set:', auc_train['Boosting'])
print('AUC for test set:', roc_auc_score(test_Y, pred))
```

```
For Boosting:
Cross-validated error rate: 0.19444444444444442
Test error rate: 0.21095334685598377
AUC for training set: 0.9138244903593945
AUC for test set: 0.8765971532605098
```

As the above code chunks show, the test error rate for the best model, Boosting, was 0.21, showing a slight increase from the cross-validated error rate of 0.19. Also, the AUC for the test set was slightly lower at 0.88 compared to the AUC for the training set 0.91. However, I do not think this is a significant difference and believe the model is generalizing to the test set quite well.

Let's look at other models to see if there is a model that performs better than Boosting in the test set:

```
[20]: test_err = {}
      models['Elastic net regression'] = elnet_class
      for name, mod in models.items():
          test_err[name] = 1 - sum(mod.predict(test_X) == test_Y) / len(test_Y)
      print('Test error rates for all models:')
      test_err
```

```
Test error rates for all models:
```

```
[20]: {'Logistic Regression': 0.2210953346855984,
       'Naive Bayes': 0.27180527383367137,
       'Elastic net regression': 0.28600405679513186,
       'Decision tree (CART)': 0.231237322515213,
       'Bagging': 0.19269776876267752,
       'Random forest': 0.21298174442190665,
       'Boosting': 0.21095334685598377}
```

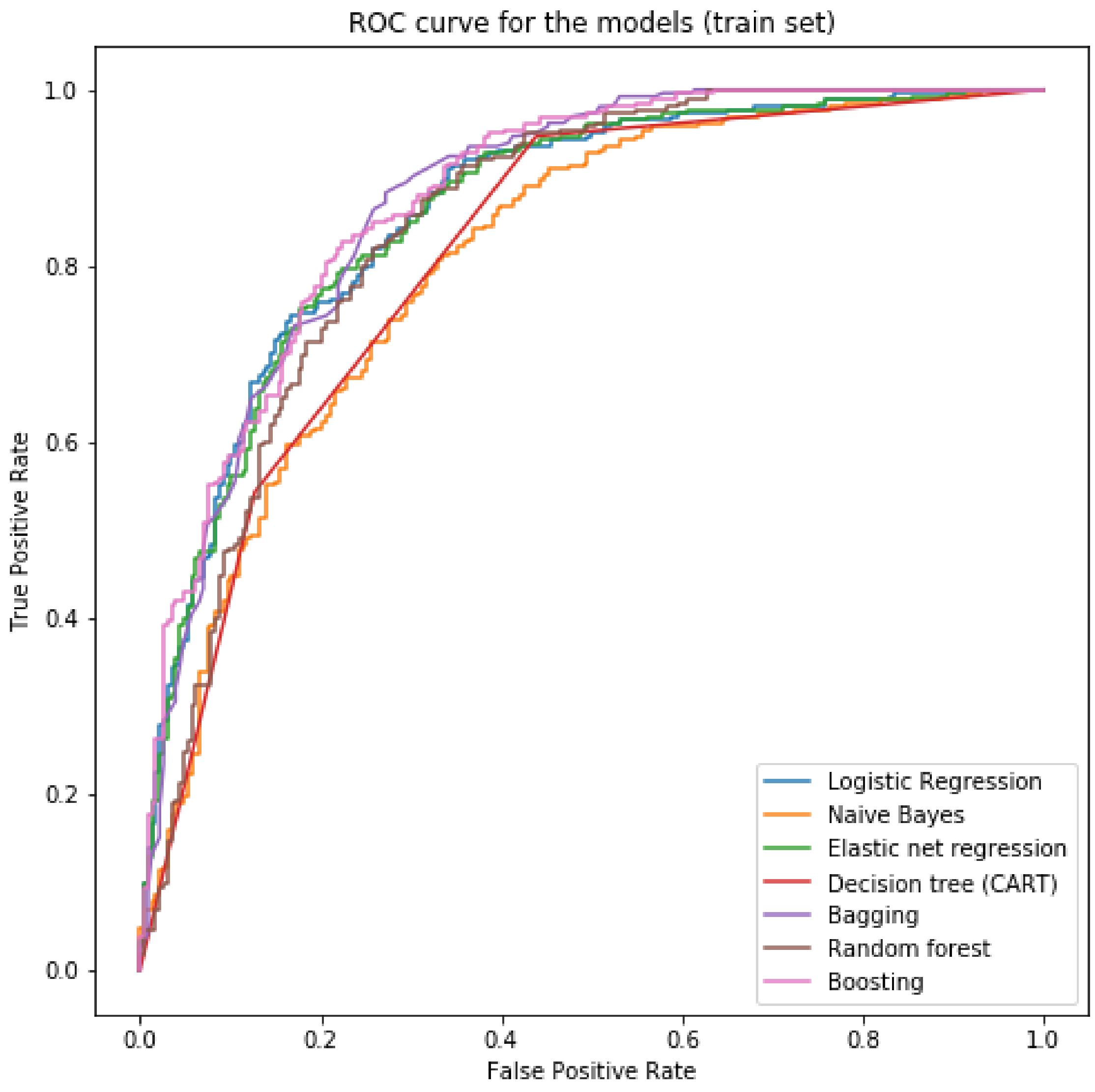
```
[21]: print('Cross-validated error rates for all models:')
      cv_err
```

```
Cross-validated error rates for all models:
```

```
[21]: {'Logistic Regression': 0.20731707317073167,
       'Naive Bayes': 0.26558265582655827,
       'Elastic net regression': 0.2554857041673917,
       'Decision tree (CART)': 0.22831978319783197,
       'Bagging': 0.19986449864498645,
       'Random forest': 0.21138211382113825,
       'Boosting': 0.19444444444444442}
```

```
[22]: models = {'Logistic Regression': lr_gridcv,
                'Naive Bayes': nb_gridcv,
                'Elastic net regression': elnet_gridcv,
                'Decision tree (CART)': cart_gridcv,
                'Bagging': bagg_gridcv,
                'Random forest': rf_gridcv,
                'Boosting': bst_gridcv}
      auc_test = {}
      plt.figure(figsize=(8,8))
      for name, mod in models.items():
          if name == 'Elastic net regression':
              pred = mod.predict(test_X)
          else:
```

```
        pred = mod.predict_proba(test_X)[:, 1]
    auc_test[name] = roc_auc_score(test_Y, pred)
    fp, tp, th = roc_curve(test_Y, pred)
    plt.plot(fp, tp, label=name)
plt.legend()
plt.title('ROC curve for the models (train set)')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate');
```



[23]:
```
print('The AUC for training set for all models:')
auc_train
```

```
The AUC for training set for all models:
```

```
[23]: {'Logistic Regression': 0.8958943444848373,
       'Naive Bayes': 0.816411577930146,
       'Elastic net regression': 0.8914233123188071,
       'Decision tree (CART)': 0.8225309465740187,
       'Bagging': 1.0,
       'Random forest': 0.8727292807510009,
       'Boosting': 0.9138244903593945}
```

```
[24]: print('The AUC for test set for all models:')
      auc_test
```

```
The AUC for test set for all models:
```

```
[24]: {'Logistic Regression': 0.8610890433631248,
       'Naive Bayes': 0.8060741476332342,
       'Elastic net regression': 0.8595829195630587,
       'Decision tree (CART)': 0.8132158225753061,
       'Bagging': 0.8721367096987752,
       'Random forest': 0.8453657729228732,
       'Boosting': 0.8765971532605098}
```

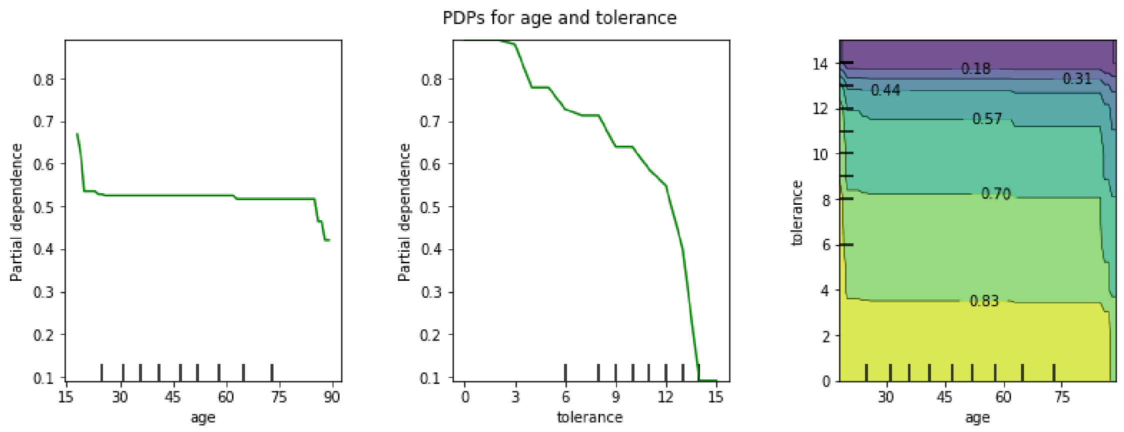
We can see that for all models, the difference between the cross-validated error rate & the test error rate, the training set AUC & the testing set AUC are generally not huge. Boosting seems to perform very well among the other models, showing the second-best error rate and best AUC. The fact that Boosting performs well relative to other models could be another evidence that the "best" model generalizes well.

Another thing to note is that the unrealistic AUC on the training set of the Bagging model showed a quite large decrease in the test set. This is proof that the Bagging model was overfitting in the training set. I do not think it is a big problem since Bagging performs very well on the test set showing the best error rate and second-best AUC, but it shows that we always have to be careful about overfitting.

The final thing to note is that Logistic Regression performs very well in the test set, too. Since logistic regression is very computationally inexpensive compared to other models we discussed, choosing logistic regression to tackle this classification problem might not be a bad idea also.

6 Present and substantively interpret the "best" model (selected in question 4) using PDPs/ICE curves over the range of: tolerance and age. Note, interpretation must be more than simple presentation of plots/curves. You must sufficiently describe the changes in probability estimates over the range of these two features. You may earn up to 5 extra points, where partial credit is possible if the solution is insufficient along some dimension (e.g., technically/code, interpretation, visual presentation, etc.).

[25]:
```
fig, ax = plt.subplots(figsize=(12,6))
plot_partial_dependence(bst_gridcv, train_X, ['age', 'tolerance', ('age',␣
↪'tolerance')],
                        feature_names=list(train_X.columns), n_jobs=4, fig=fig)
fig.suptitle('PDPs for age and tolerance', y=0.75);
```



Since the best model I chose is Boosting (Gradient Boosting), which is quite hard to interpret, I will need help from PDP to interpret what is going on in the model. Let's discuss the one variable PDPs first. On the leftmost plot, we can clearly see that `age` is not affecting the target feature `colrac` that much. The partial dependence stays almost the same as the age increases. There is a slope at both ends of the plot, but this is probably because the data is sparse in the edges. I would conclude that age is, quite counter-intuitively, has low feature importance in the model.

On the contrary, we could see a clear pattern on the second one variable PDP in the center. As `tolerance` increases, partial dependence decreases. This means that `tolerance` is negatively related to `colrac`. We could see that the slope becomes steeper as the `tolerance` gets bigger, suggesting that `tolerance` has more effect on `colrac` as `tolerance` becomes larger.

The two-variable PDP on the right shows the interaction between `age` and `tolerance`. We can see that the pattern of `age` and `tolerance` follows the pattern shown in the one variable PDP, suggesting there is no interaction between the two in terms of feature importance. Age has almost no effect on `colrac` even in interaction with `tolerance`, and `tolerance` keeps the pattern of negatively effecting `colrac` as it becomes larger. Also, the feature importance of `tolerance` getting bigger as it increases is shown by the narrower gap between the gradient line (same with the one variable PDP).

To conclude, `age` is not an important feature in the Boosting model, and `tolerance` seems to be negatively affecting `colrac`, with having stronger effects as the `tolerance` gets higher. There doesn't seem to be a significant interaction between the two features in terms of feature importance.