Homework 4

Instructions. This homework is due by Friday, September 29th at 5pm. You can edit this file and copy/paste into it by opening it in Libreoffice on Linux, PDFfiller Google App, Acrobat, Microsoft Word and others. If you use the internet for information, please assess the verity of the source and document it as a source. Please turn in a pdf version of the written solutions, as well as a zip file of all code you use to solve the problems. Please name each piece of code by the question it is intended to answer or otherwise clearly indicate to me how to run the code to get each answer.

1. There are two files on Blackboard and `hpc-class`, `yield.train.csv` and `yield.test.csv`, that contain information on crop yields and various potential predictors (features), as listed below:

    1. County name
    2. Year (1995 – 2004)
    3. Drought Index (DI)
        - Normal conditions (0)
        - Wet conditions (> 0)
        - Drought conditions (< 0)
    4. Yield: the response variable
        - Very Low
        - Low
        - High
        - Very High
    5. Soil: the quality of the soil, higher is better
    6. Rainfall (RF): rainfal recorded for the period May–Sep (tenths of millimeters)
    7. TMin, TMax: average mininum or maximum temperature recorded for the period May–Sep (tenths of Celsius)

    (a) Train a decision tree using `sklearn.tree.DecisionTreeClassifier` using the training data and assessing performance on the test data. Assess the performance using Matthew's correlation coefficient, accessible via `sklearn.metrics.matthews_corrcoef()`.

    i. The Gini impurity index for a group of observations in a multi-class classification problems is

$$G = \sum_{i=1}^{n_c} p_i(1 - p_i),$$

    where n_c is the number of classes, and p_i is the proportion of the observations in the ith class. The importance of split s is

$$I_s = G_{s,\text{parent}} - G_{s,\text{child 1}} - G_{s,\text{child 2}},$$

where $G_{s,\text{parent}}$ is the Gini index of the parent node, and the others are the Gini indices of the daughter nodes. The **Gini importance** of a feature is obtained by averaging the importance of all splits involving that feature, possibly weighting by the number of affected observations. What feature has the largest Gini importance in the tree?
(Hint: see the sklearn.tree.DecisionTreeClassifier manual).

Solution: I found it painful to manipulate the dataset, and I learned a lot of Python to get this done. Python's abundance of libraries is a blessing and a curse: a blessing because there is code to help you do most things and a curse because there are so many ways to accomplish the same task.

Reading data. First, I decided to use Python's `argparse` library to get the training and test data files from the command line. Then, I used Python's `csv` library to parse the training and test data files, which are in csv format. Note, I dropped the first (index 0) column, which is just the row number. I also parsed the fourth (index 3) column differently, since it is the response variable. In the end, I end up with the predictors (features) in `training_x` or `test_x` and the response in `training_y` or `test_y`.

Converting data. Now we need to convert all the predictors to floats. For this purpose, I use Python's `panda` library. I convert the combined data to a `DataFrame`, asking it to attempt conversion to float, which works for Year but not for County. To convert the string predictor "County" to float, I use `numpy`'s `unique()` to convert `County` to `integer`, which `DecisionTreeClassifer()` appears able to handle.

Fitting decision trees. Next, I instantiate a `DecisionTreeClassifier`. I can fit it with `DecisionTreeClassifier.fit()`, passing in the converted training features and the response, which need not be converted. However, there is randomness in the fitting process, so each call will produce a different fitted tree. To determine whether the differences I observe are real, I will repeat the fitting `n_repeat`=1,000 times and compute the average importance of each feature.

Importance. Upon reading the manual for the `DecisionTreeClassifier`, I learn that attribute `feature_importances_` of the `DecisionTreeClassifier` object holds the Gini importances. Thus, I will compute the average (across replicate runs) Gini importance for each variable and report the feature with the highest average importance. The output of command
`python hw4q1a.py -q -r yield.train.csv -e yield.test.csv`
is shown below. The code will be reserved until the end of this part.

```
## Year
```

Thus, feature `Year` is the most important variable for predicting yield.

Extra. Because the fitted object is random, all statistics computed from that object, such as Gini importance, are random variables. However, we want to draw conclusions about the *population importance*, a quantity we cannot directly observe. Extra code (and output without the `-q` option) demonstrates how you can be sure (or almost sure) that `Year` and the other features are truly important. We find that some variables, such as `TMin.July.`, are not important. In addition, we learn that `Year` is significantly more important than the next important variable `Soil..WA.`, the soil quality. Given what little I know about farming, these features seem like they should be important variables for predicting yield.

ii. What maximum tree depth achieves the best performance? Beware of randomness in the answer!

Solution: Some investigation reveals that the `max_depth` argument to the `DecisionTreeClassifer` constructor determines the maximum tree depth. After some manual investigation, I decide to try tree depths between 1 and 10. To measure the performance of the tree, I predict the test data and compute Matthew's Correlation Coefficient (MCC) using function `sklearn.metrics.matthews_corrcoef()`. As before, I repeat the fitting to account for noise in the estimated MCC. Because of the time it takes, I only repeat the fitting `n_repeat`= 100 times. The maximum depth yielding best performance is given by the output of command that requests output for part ii:
`python hw4q1a.py -q -r yield.train.csv -e yield.test.csv -p ii`
as

```
## 8
```

Extra. As before, I can perform a t-test to see if `max_depth=8` is significantly better, as measured by MCC, than the next best depth, `max_depth=6`. On most runs, I find that `max_depth=8` does yields a significantly better MCC. The default `max_depth` is unlimited, and I also find that restricting `max_depth=8` is better than using the default.

iii. What causes the randomness in the estimated trees from multiple runs on the same data?

Solution: The manual states "The features are always randomly permuted

at each split. Therefore, the best found split may vary, even with the same training data and max_features=n_features, if the improvement of the criterion is identical for several splits enumerated during the search of the best split." Therefore, if two features are equally good at predicting the outcome at a particular node in the tree, then which one is chosen will vary from run to run. In this dataset on crop yield, there was detectable variability, but the variance was very small, and hence even small differences, in importance or accuracy, were significantly different.
The code from this part is shown below.

```
from sklearn import tree, metrics
import scipy as sp, pandas as pd, numpy as np, sklearn as sl
import csv, sys, math
import argparse

# parse command line arguments
parser = argparse.ArgumentParser()
parser.add_argument("-r", "--training",
   default = "yield.train.csv",
   help = "file with training data")
parser.add_argument("-e", "--testing",
   default = "yield.test.csv",
   help = "file with testing data")
parser.add_argument("-p", "--part", default = 'i',
   choices = ['i','ii','iii','iv','graph'])
parser.add_argument("-q", "--quiet", action = "store_true")
args = parser.parse_args()

training_file = args.training
testing_file = args.testing

# yield responses in order
yield_names = ["Very Low", "Low", "High", "Very High"]

### preparing the yield data
## data input
# read in training data and record feature names
feature_names = []   # names of features
training_x = []      # to store training features
training_y = []      # to store training response
with open(training_file, 'r') as csvfile:
   yield_reader = csv.reader(csvfile, delimiter=',',
      quotechar='"')
```

```
   i = 1
   for row in yield_reader:
      del row[0]  # remove column number
      if i==1:
         del row[3]  # remove response
         feature_names = row
      else:
         training_y.append(row[3])
         del row[3]  # remove response
         training_x.append(row)
      i += 1

# size of training data
n_training = len(training_x)

# read in testing data, ignoring first line
testing_x = [] # to store testing features
testing_y = [] # to store testing response
with open(testing_file, 'r') as csvfile:
   yield_reader = csv.reader(csvfile, delimiter=',',
      quotechar='"')
   i = 1
   for row in yield_reader:
      del row[0]
      if i != 1:
         testing_y.append(row[3])
         del row[3]  # remove response
         testing_x.append(row)
      i += 1

# size of testing data
n_testing = len(testing_x)

## end data input

## massage data to convert all features to floats
# convert to pandas' DataFrame and request conversion to float
# combine data so conversion is consistent across testing and
# training
df = pd.DataFrame(training_x + testing_x,
   columns = feature_names, dtype = "double")
```

```
# convert counties to numeric, then float
counties, df['County'] = np.unique(df['County'],
   return_inverse = True)

# resplit features into training and testing
training_x = df[:n_training]
testing_x = df[n_training:]

## end massage data
### end preparing yield data

## run decision tree classifier
def main():

   if args.part == 'i':
      if args.quiet == False:
         print "Part 1(a)i"

      # prepare classifier
      dtc = tree.DecisionTreeClassifier()

      # I will repeat the calculations to deal with
      # the variability in the estimated trees.
      gini_imp_avg = [0] * len(feature_names)
      gini_imp_sd = [0] * len(feature_names)
      n_repeat = 1000
      for i in range(n_repeat):
         # fit classifier
         dtc.fit(training_x, training_y)

         # friendly output showing progress...
         if args.quiet == False:
            sys.stdout.write(".")
            sys.stdout.flush()
            if (i+1) % 50 == 0:
               print ' {0:4d}/{1:4d}'.format(i+1, n_repeat)

         # get importances, computing mean and standard deviation
         for j in range(len(feature_names)):
            gini_imp_avg[j] += dtc.feature_importances_[j]
            # This is a dangerous way to compute standard
            # deviation (prone to overflow), but it is
```

```
            # more efficient.
            # Correct way: compute mean and then compute mean
            # squared deviations from mean.
            gini_imp_sd[j] += dtc.feature_importances_[j]**2

      # Now I can use a t-test to explore H0: mu = 0 and
      # decide when an importance is significantly non-zero.
      gini_imp_p_values = []
      for j in range(len(feature_names)):
         gini_imp_avg[j] /= n_repeat
         gini_imp_sd[j] = math.sqrt(gini_imp_sd[j]/n_repeat
            - gini_imp_avg[j]**2)
         gini_imp_p_values.append(2*sp.stats.t.cdf(
            -math.fabs(gini_imp_avg[j])/gini_imp_sd[j],
            df = n_repeat - 1))

      # Report the feature with largest gini importance.
      if args.quiet == False:
         sys.stdout.write("The largest gini importance" +\
            " belongs to feature")
      print feature_names[gini_imp_avg.index(max(gini_imp_avg))]
      if args.quiet == False:
         print "\n\nEXTRA INFORMATION:\n"
         print "Gini importance for feature names:",\
            feature_names
         print "Average:", gini_imp_avg
         print "Standard deviation:", gini_imp_sd
         print "P-values (H0: mu = 0):", gini_imp_p_values

         # Most are significant, but the first and third are
         # the topmost significance.  A two-sample t-test with
         # unequal variance allows us to test H0: \mu_1 = \mu_3.
         t_stat = (gini_imp_avg[0] - gini_imp_avg[2])\
            / math.sqrt((gini_imp_sd[0]**2
            + gini_imp_sd[2]**2)/n_repeat)
         df = ((gini_imp_sd[0]**2
            + gini_imp_sd[2]**2)/n_repeat)**2\
            / ((gini_imp_sd[0]**2/n_repeat)**2
            + (gini_imp_sd[2]**2/n_repeat)**2)*(n_repeat - 1)
         print "P-value (H0: mu1 = mu3):", 2*sp.stats.t.cdf(
            -math.fabs(t_stat), df = df)
   elif args.part == 'ii':
```

```
if args.quiet == False:
   print "Part 1(a)ii"
   print "Below, max_depth=0 implies no limit on depth"

max_depth = 10
n_repeat = 100
mcc_avg = [0] * (max_depth+1)
mcc_sd = [0] * (max_depth+1)
for depth in range(max_depth+1):
   if depth > 0:
      dtc = tree.DecisionTreeClassifier(max_depth = depth)
   else:
      dtc = tree.DecisionTreeClassifier()
   for i in range(n_repeat):
      if args.quiet == False:
         sys.stdout.write(".")
         sys.stdout.flush()
         if (i+1) % 50 == 0:
            print ' {0:4d}/{1:4d}'.format(i+1, n_repeat)
      # fit tree
      dtc.fit(training_x, training_y)

      # predict results
      predicted_y = dtc.predict(testing_x)
      mcc = sl.metrics.matthews_corrcoef(testing_y,
         predicted_y)
      mcc_avg[depth] += mcc
      mcc_sd[depth] += mcc**2
   mcc_avg[depth] /= n_repeat
   mcc_sd[depth] = math.sqrt(
      mcc_sd[depth]/n_repeat - mcc_avg[depth]**2)
   if args.quiet == False:
      print "Max. depth " + str(depth) + " mcc: "\
         + str(mcc_avg[depth]) + " +/- "\
         + str(mcc_sd[depth])
if args.quiet == False:
   sys.stdout.write("The max. depth with highest mean" +\
      " accuracy is ")
print range(max_depth+1)[mcc_avg.index(max(mcc_avg))]
if args.quiet == False:
   idx = [6, 8]
   t_stat = (mcc_avg[idx[0]] - mcc_avg[idx[1]]) /\
```

```
            math.sqrt((mcc_sd[idx[0]]**2\
            + mcc_sd[idx[1]]**2)/n_repeat)
         df = ((mcc_sd[idx[0]]**2 + mcc_sd[idx[1]]**2) /\
            n_repeat)**2 / ((mcc_sd[idx[0]]**2/n_repeat)**2\
            + (mcc_sd[idx[1]]**2/n_repeat)**2)\
            * (n_repeat - 1)
         print "P-value (H0: mu" + str(idx[0]) + " == mu" +\
            str(idx[1]) + "):  " +\
            str(2*sp.stats.t.cdf(-math.fabs(t_stat), df = df))
         idx = [0, 8]
         t_stat = (mcc_avg[idx[0]] - mcc_avg[idx[1]]) /\
            math.sqrt((mcc_sd[idx[0]]**2\
            + mcc_sd[idx[1]]**2)/n_repeat)
         df = ((mcc_sd[idx[0]]**2 + mcc_sd[idx[1]]**2)/\
            n_repeat)**2 / ((mcc_sd[idx[0]]**2/n_repeat)**2\
            + (mcc_sd[idx[1]]**2/n_repeat)**2)\
            * (n_repeat - 1)
         print "P-value (H0: mu" + str(idx[0]) + " == mu" +\
            str(idx[1]) + "):  " +\
            str(2*sp.stats.t.cdf(-math.fabs(t_stat), df = df))
   elif args.part == 'graph':
      # using best performer
      dtc = tree.DecisionTreeClassifier(max_depth = 8)
      dtc.fit(training_x, training_y)

      # prepare plot as per scikit tutorial
      import graphviz

      dot_data = tree.export_graphviz(dtc, out_file = None,
         feature_names = feature_names, class_names = training_y,
         filled = True, rounded = True, special_characters = True)
      tree.export_graphviz(dtc, out_file = "test.dot",
         feature_names = feature_names, class_names = training_y,
         filled = True, rounded = True, special_characters = True)
      graph = graphviz.Source(dot_data)
      graph.render("yield")
   else:
      print "ERROR: --part iv not valid for part (a)"

if __name__ == "__main__":
   main()
```

(b) Train a random forest using `sklearn.ensemble.RandomForestClassifier` using

the training data and assessing performance on the test data.

i. Why does increasing the `n_estimators` argument reduce the variability in the estimated random forests from multiple runs?

Solution: I repeatedly (`n_repeat`= 100 times) estimate the random forest for various choices of `n_estimators` in the set $\{10, 20, 30, 40, 50, 75, 100, 250, 500\}$ and compute the MCC. Because the runs get quite slow, I take advantage of the fact that `sklearn.ensemble.RandomForestClassifier()` takes an argument `n_jobs` that allows us to use more cores. To measure variability in the estimated random forests, I use Matthew's Correlation Coefficient (MCC) as a summary statistic on the result and compute the sample standard deviation of the MCC . Running command
`python hw4q1b.py -q -r yield.train.csv -e yield.test.csv`
yields:

```
## MCC for n_esimators =   10  is 0.563  +/- 0.041
## MCC for n_esimators =   20  is 0.600  +/- 0.039
## MCC for n_esimators =   30  is 0.616  +/- 0.033
## MCC for n_esimators =   40  is 0.615  +/- 0.027
## MCC for n_esimators =   50  is 0.615  +/- 0.026
## MCC for n_esimators =   75  is 0.629  +/- 0.020
## MCC for n_esimators =  100  is 0.626  +/- 0.020
## MCC for n_esimators =  250  is 0.629  +/- 0.017
## MCC for n_esimators =  500  is 0.630  +/- 0.012
```

As $m :=$ `n_estimators` increases, we see the standard deviation gradually declines. Because the random forest combines the prediction of m trees, it benefits from the Law of Large Numbers (LLN). Specifically, if $Y_i \in \{\text{Very Low}, \text{Low}, \text{High}, \text{Very High}\}$ is the prediction of the ith tree in the random forest, then the prediction of the random forest is

$$\text{argmax}_{x \in \{\text{Very Low,Low,High,Very High}\}} \sum_{i=1}^{m} \mathbb{1}\{Y_i = x\}.$$

If we divide this count by m, then we are maximizing over

$$\hat{p}_j = \frac{1}{m} \sum_{i=1}^{m} \mathbb{1}\{Y_i = x\},$$

for $j \in \{\text{Very Low}, \text{Low}, \text{High}, \text{Very High}\}$. The LLN tells us that $\hat{p}_j \to p_j$, where p_j the probability an infinitely large random forest will predict class j. In particular, the chance that $\hat{p}_j$ will be greater than a certain ϵ distance from p_j grows infinitesimally small as m increases. For large enough m, the random forest will be effectively making predictions based on p_j, which contain no estimation error, and thus the random forest will become deterministic (nonrandom) in this limit.

ii. How does restricting the maximum depth of the tree affect performance? Thoughts why?

Solution:

Affect on performance. The performance also appears to increase with `n_estimators`, though it appears to be plateauing, thus we use `n_estimators` = 1000 to answer this question and hope the variability will be reduced enough to detect which depth offers the best performance.

```
## MCC for max depth =  0 is 0.607
## MCC for max depth =  1 is 0.217
## MCC for max depth =  2 is 0.350
## MCC for max depth =  3 is 0.469
## MCC for max depth =  4 is 0.539
## MCC for max depth =  5 is 0.544
## MCC for max depth =  6 is 0.585
## MCC for max depth =  7 is 0.600
## MCC for max depth =  8 is 0.614
## MCC for max depth =  9 is 0.614
## MCC for max depth = 10 is 0.602
```

It is pretty clear that the performance increases as we up the maximum depth. It may appear that there is a best performance for `max_depth=8` or so, which enticingly agrees with the results for the Decision Tree. Unfortunately, this is not a repeatable finding, and most runs I did found that placing no depth on the trees (the default) yields the best prediction. One could, of course, do statistical tests to back up this conclusion, but it is quite costly (in time) to do so.

Thoughts why. The reason limiting the depth is helpful for Decision Trees is because Decision Trees have a real risk of overfitting the data. Thus, limiting the maximum depth estimates a less complex model (less splits, less decisions, therefore less complex), which performs better on the test data. The same reasoning does not apply to random forests, which resample the data with replacement for each tree in the forest and then combine the predictions of all trees. While it is true that any one tree may be overfit to the data, the random forest averages out those tendencies by combining the predictions of all trees. Specifically, an overfit tree creates a bias in the prediction such that it is more (or less likely) to predict a certain outcome than it should. It is very unlikely that the bias will be in the same direction in all trees of the forest, so the forest prediction does not suffer from this bias.

iii. How does the fraction of features considered at each split affect performance?

Thoughs why?

Solution:

Affect on performance. I varied `n_features` in $\{0.1, 0.2, \ldots, 0.9, 1.0\}$ with `n_estimators` = 1000. Repeated runs showed that there was considerable variation in performance, as measured by MCC, on par with the variation across choices of `n_features`. This makes it hard to know whether there are actual differences. Once can repeat the experiment many times and perform t-tests, as we did for Part (a), but again, it would take a long time. Instead, I manually ran the code 10 times. The argument `n_features` $\notin \{0.1, 1.0\}$ for those 10 runs, suggesting (but p-value only around 0.1: the probability of at most 0 successes in 10 trials with probability of success 0.20) that performance is better for intermediate values of `n_features`. Here is one set of output upon running command
`python hw4q1b.py -q -r yield.train.csv -e yield.test.csv -p iii`

```
## MCC for max_features = 0.10 is 0.612
## MCC for max_features = 0.20 is 0.592
## MCC for max_features = 0.30 is 0.630
## MCC for max_features = 0.40 is 0.630
## MCC for max_features = 0.50 is 0.679
## MCC for max_features = 0.60 is 0.643
## MCC for max_features = 0.70 is 0.620
## MCC for max_features = 0.80 is 0.633
## MCC for max_features = 0.90 is 0.636
## MCC for max_features = 1.00 is 0.623
```

iv. Compute and display the Confusion matrix from the best random forest result you have obtained. Does the classifier have particular problem distinguishing certain outcomes and not others? Why might that be?

Solution: I discover that `pandas` knows how to compute confusion matrices, with the `pandas.crosstab()` command, so using `n_estimators` = 1000, `n_jobs` = 8 and all other default parameters as per what we have learned, I find the following confusion matrix.

```
## preds      Very Low  Low  High  Very High
## actual
## Very Low         12    7     1          0
## Low               4   64     8          0
## High              0    6    29          1
## Very High         0    0     5          4
```

Code. The code for this part is given below.

```
from hw4q1a import *
from sklearn import ensemble

if args.part == 'i':
   n_repeats = 100
   n_estimators_list = [10, 20, 30, 40, 50, 75, 100, 250, 500]
   for n_estimators in n_estimators_list:
      rfc = ensemble.RandomForestClassifier(n_estimators
         = n_estimators, n_jobs = 8)
      mcc_avg = 0
      mcc_sd = 0
      for i in range(n_repeats):
         rfc.fit(training_x, training_y)
         predicted_y = rfc.predict(testing_x)
         mcc = metrics.matthews_corrcoef(\
            testing_y, predicted_y)
         mcc_avg += mcc
         mcc_sd += mcc**2

      mcc_avg /= n_repeats
      mcc_sd = math.sqrt(mcc_sd/n_repeats - mcc_avg**2)
      print 'MCC for n_esimators = {0:4d}'.format(n_estimators),\
         ' is {0:.3f}'.format(mcc_avg),\
         ' +/- {0:.3f}'.format(mcc_sd)

if args.part == 'ii':
   n_estimators = 1000
   max_max_depth = 10
   for max_depth in range(max_max_depth+1):
      if max_depth == 0:
         rfc = ensemble.RandomForestClassifier(
            n_estimators = n_estimators, n_jobs = 8)
      else:
         rfc = ensemble.RandomForestClassifier(
            n_estimators = n_estimators, n_jobs = 8,
            max_depth = max_depth)
      rfc.fit(training_x, training_y)
      predicted_y = rfc.predict(testing_x)
      mcc = metrics.matthews_corrcoef(testing_y, predicted_y)
      print 'MCC for max_depth = {0:2d} is {1:.3f}'.format(\
         max_depth, mcc)
```

```
if args.part == 'iii':
   n_estimators = 1000
   for max_features in range(10, 110, 10):
      rfc = ensemble.RandomForestClassifier(
         n_estimators = n_estimators, n_jobs = 8,
         max_features = max_features/float(100))
      rfc.fit(training_x, training_y)
      predicted_y = rfc.predict(testing_x)
      mcc = metrics.matthews_corrcoef(testing_y, predicted_y)
      print 'MCC for max_features = {0:.2f} is {1:.3f}'.format(\
         max_features/float(100), mcc)


if args.part == 'iv':
   n_estimators = 1000
   rfc = ensemble.RandomForestClassifier(
      n_estimators = n_estimators, n_jobs = 8)
   rfc.fit(training_x, training_y)
   predicted_y = rfc.predict(testing_x)
   for_ct_testing_y = pd.Categorical(testing_y,\
      categories = yield_names, ordered = True)
   for_ct_predicted_y = pd.Categorical(predicted_y,\
      categories = yield_names, ordered = True)
   print pd.crosstab(for_ct_testing_y, for_ct_predicted_y,\
      rownames=['actual'], colnames=['preds'])
```

2. Apply the decision tree or random forest to the permissive, nonpermissive virus sequence data to determine (1) if you can distinguish permissive and nonpermissive cells based only on the virus fragments they contain and (2) what motifs (nucleotide patterns) best distinguish the two types of cells. This is an open-ended question. You must identify useful features, choose on of your two machine learning techniques, and choose its parameter settings. Justify and critique your choices. Analyze and interpret your results.

Solution:

Methods. I decide to use a random forest because it had better performance as measured by Matthew's Correlation Coefficient in Question 1. Furthermore, it does not require that I optimize over maximum depth. I do not have much data, so I

randomly sample 25% of the sequences from each of permissive and nonpermissive cells to use as test data. For all the data, I decide to use as features the proportion of mononucleotides and dinucleotides in each sequence. To assess whether I can distinguish permissive and nonpermissive cells, I examine MCC on the test data. To assess which features best distinguish the two types, I examine the Gini importances. Here is the output (the code is appended to the end of this solution).

```
## Matthew's correlation coefficient: 0.845154254729
##    Feature      Gini
## 18      TG  0.162892
## 14      GG  0.131189
## 16      TA  0.103601
## 12      GA  0.080286
## 6       AG  0.071119
## 9       CC  0.059069
## 0        A  0.055124
## 2        G  0.048585
## 15      GT  0.043807
## 5       AC  0.036794
## 13      GC  0.033944
## 4       AA  0.024610
## 3        T  0.023512
## 17      TC  0.022807
## 1        C  0.019734
## 7       AT  0.019008
## 10      CG  0.018286
## 11      CT  0.016666
## 19      TT  0.015262
## 8       CA  0.013705
```

Results. The results, especially in terms of performance, are highly variable across replicate runs, as the training and testing data partition changes. Machine learning techniques work best on large datasets, and our dataset is quite small. With such small data, one can enhance performance by inputing information about relevant features sourced from independent information. Since you do not have independent information (knowledge from HW3 was based on the same data), you have little opportunity to improve the performance without risking overfitting to the current dataset.

There was more consistency in the most important features. In particular, dinucleotides TG, GG, and TA were always at the top. This may suggest that motif TG is targeted to TA. It is less clear to what GG is mutated. Either AG or GA

and even AA vie for some level of importance. Among the mononucleotides, A and G were most important, which agrees with our previous discovery that G $\rightarrow$ A. However, since dinucleotides were more predictive of outcome, we suspect that the mutaton *does* target a motif rather than independently attacking Gs. And perhaps dinucleotide TG is one of those motifs.

Code.

```
# We'll use code from last week's homework!
from hw3q2biii import perm_fasta, nonperm_fasta
import pandas as pd, sys, random as rnd
from sklearn import ensemble, metrics

nucleotides = ['A','C','G','T']  # the 4 nucleotides
dinucleotides = []      # the 16 dinucleotides...

for nuc1 in nucleotides:
   for nuc2 in nucleotides:
      dinucleotides.append(nuc1 + nuc2)

# all the features we will consider
features = nucleotides[:] + dinucleotides[:]


##
# Estimate nucleotide proportions in a sequence.
#
# @param seq   sequence as string
# @return   array of proportions in same order as \par nucleotides
def nuc_proportions(seq):
   seq_len = len(seq)
   nuc_cnts = []
   for nuc in nucleotides:
      nuc_cnts.append(seq.count(nuc) / float(seq_len))
   return nuc_cnts

##
# Count dinucleotides in a sequence.
#
# @param seq   sequence as string
# @return   array of proportions in same order as \par dinucleotides
def dinuc_proportions(seq):
```

```
   seq_len = len(seq) - 1
   dinuc_cnts = [0] * len(dinucleotides)
   for j in range(1, len(seq)):
      dinuc = seq[j-1] + seq[j]
      dinuc_cnts[dinucleotides.index(dinuc)] += 1
   for j in range(len(dinucleotides)):
      dinuc_cnts[j] /= float(seq_len)
   return dinuc_cnts



# build dataframe of features & response: Permissive or Nonpermissive
df = pd.DataFrame(columns = ["Permissive"] + features[:])

i = 0
# permissive data
for name in perm_fasta:
   seq = perm_fasta[name]
   df.loc[i] = ['Permissive'] + nuc_proportions(seq)\
      + dinuc_proportions(seq)
   i += 1

n_permissive = i

# nonpermissive data
for name in nonperm_fasta:
   seq = nonperm_fasta[name]
   df.loc[i] = ['Nonpermissive'] + nuc_proportions(seq)\
      + dinuc_proportions(seq)
   i += 1

n_nonpermissive = i - n_permissive

# sample 25% of permissive/nonpermissive for testing
test_idx = rnd.sample(range(n_permissive), k = int(0.25*n_permissive))
test_idx = test_idx + rnd.sample(range(n_permissive, i),
   k = int(0.25*n_nonpermissive))

# separate training, testing, features, response
x = df[features]
y = df['Permissive']
training_x = x[~df.index.isin(test_idx)]
```

```
testing_x = x[df.index.isin(test_idx)]
training_y = y[~df.index.isin(test_idx)]
testing_y = y[df.index.isin(test_idx)]

# build random forest
rfc = ensemble.RandomForestClassifier(n_estimators = 1000, n_jobs = 8)
rfc.fit(training_x, training_y)
predicted_y = rfc.predict(testing_x)

# assess performance
mcc = metrics.matthews_corrcoef(testing_y, predicted_y)
print "Matthew's correlation coefficient:", mcc

# sort features by Gini importance
data_result = {"Feature": features, "Gini": rfc.feature_importances_}
df_result = pd.DataFrame(data = data_result)
print df_result.sort_values(by = "Gini", ascending = False)
```

Hint. I do not know what you learned in lab, but I found the pandas Python library particularly helpful for massaging the data. In particular, I learned a lot from this blog post.