CMPT 423/820 Assignment 2 Question 1

Model Solution and Grading Scheme

Prologue: Importing and reading the Data

In [4]:
```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

from sklearn.naive_bayes import GaussianNB
from sklearn import model_selection

# a list of names for the columns in the data
cnames=['SepalLengthCm', 'SepalWidthCm',
        'PetalLengthCm', 'PetalWidthCm', 'Species']
dataframe = pd.read_csv('iris.csv',
                        header=None,
                        names=cnames,
                        index_col=False)

# Separate the features from the class
array = dataframe.values
X = array[:,0:4]
Y = array[:,4]
```

Part 1. Four 1-Feature Classifiers

The question asks for 4 one-feature classifiers. To do this, I'll split the data into individual columns. To evaluate the classifiers, I'll use simple 10-fold cross-validation.

In [5]:
```
for column in range(4):
    # select the column
    X_col_train = X[:,column:column+1]

    # cross validation to evaluate the performance using accuracy
    kfold = 10
    cv_results = model_selection.cross_val_score(GaussianNB(), X_col_train, Y,
                                                 cv=kfold, scoring='accuracy')

    #displaying the mean and standard deviation of the prediction
    msg = "%s %s: \t%f (%f)" % ('NB accuracy for feature', cnames[column],
                                 cv_results.mean(), cv_results.std())
    print(msg)
```

```
NB accuracy for feature SepalLengthCm:  0.726667 (0.081377)
NB accuracy for feature SepalWidthCm:   0.560000 (0.067987)
NB accuracy for feature PetalLengthCm:  0.953333 (0.042687)
NB accuracy for feature PetalWidthCm:   0.953333 (0.052068)
```

Commentary: solutions and grading

Some form of evaluation on a test set is necessary. It's not good enough to compare error from the training set only. The following variations are acceptable for full marks:

- A simple test-training split of the data. This is not as good as CV, because the results will be more dependent on the particular split.
- Cross-validation with a different k, and a different proportion of the data set aside for testing.

The results will depend on the method used. The accuracies might be a bit higher or lower. The 4th column ("petal width") is expected to have the highest accuracy. The second column ("sepal width") should have the lowest accuracy.

Grading For full marks:

- Your Python scripting was neat and presentable.
- You made good use of Python comments, and Markdown cells to explain your method to a reader.
- You calculated the accuracies correctly (using cross-validation or a test-training. split of the data), and presented them neatly.

Part 2: One Four-Feature Classifier

Now we'll use all the features, repeating the evaluation with simple 10-fold cross-validation.

In [6]:
```
# cross validation to evaluate the performance using accuracy
kfold = 10
cv_results = model_selection.cross_val_score(GaussianNB(), X, Y, cv=kf
old, scoring='accuracy')

#displaying the mean and standard deviation of the prediction
msg = "%s: %f (%f)" % ('NB accuracy using all columns', cv_results.mea
n(), cv_results.std())
print(msg)
```

```
NB accuracy using all columns: 0.953333 (0.042687)
```

Commentary: solutions and grading

The accuracy should be calculated with the same methodology, preferably exactly the same details. Using the same method means the comparison is least susceptible to false conclusions.

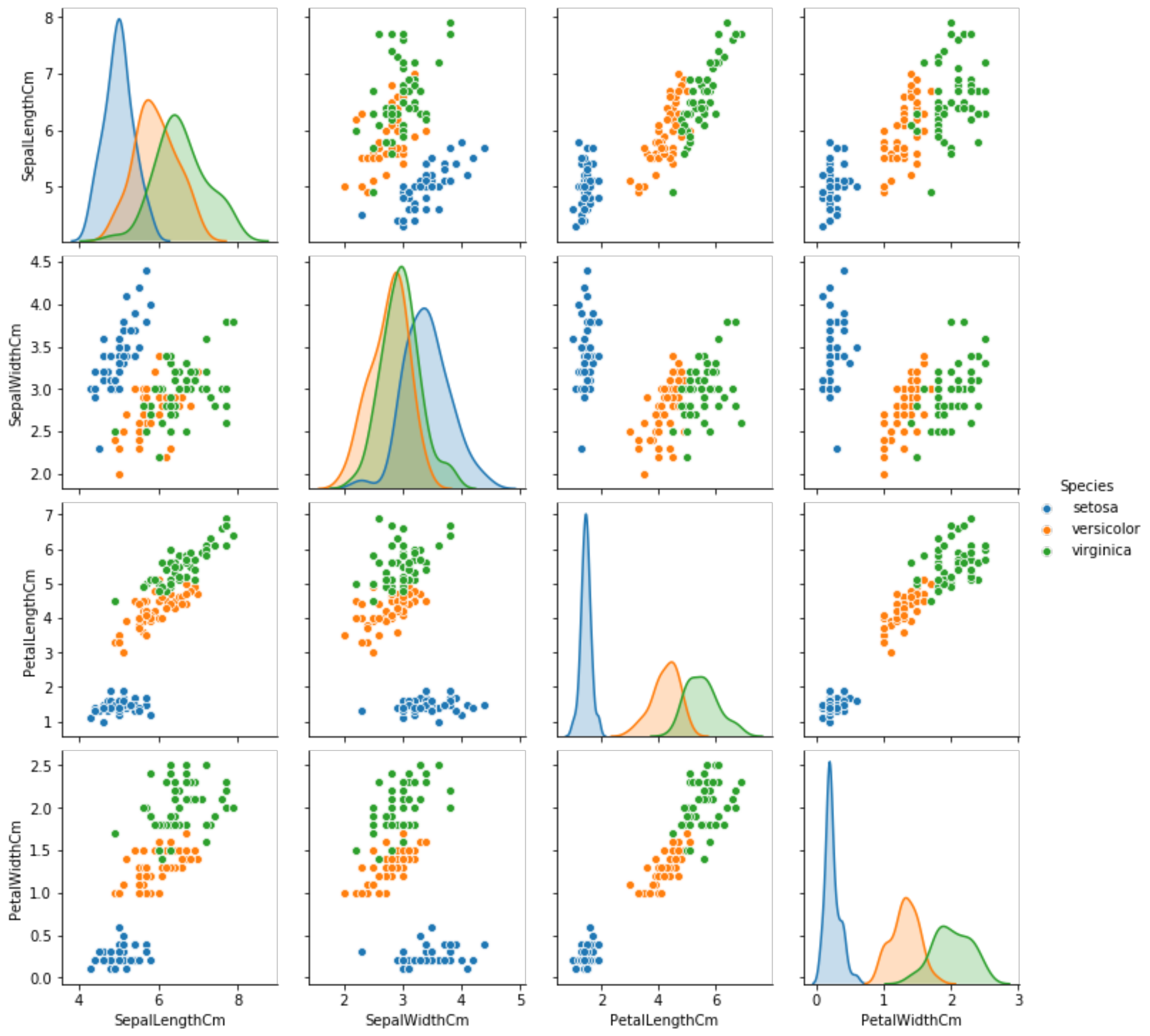
The results will depend on the method used to evaluate accuracy. A simple test-train split could make the four-feature classifier look better than the best 1-feature classifier. In any case, it should be close.

Grading This part has no grade associated with it.

Part 3: Density plots

Now we plot the density of the data. The Assignment description suggested A1Q7 Task 4, but using Seaborn seemed easier. It should have been copy-paste, anyway.

In [4]:
```python
sns.pairplot(dataframe, hue="Species", diag_kind='kde')
plt.show()
```



The density plots are above, along the diagonal. Each curve represents the distribution of the feature for a given class value (setosa, versicolor, virginica). In other words, each curve represents $P(F|C)$. While the density plots are not simple Gaussian distributions (they are a bit lumpy in places), the Gaussian Naive Bayes classifier would fit nice neat bell curves in place of these densities.

The more overlap there is between the density curves for a given feature, the less well the feature does as a single feature classifier. The best feature is Petal Width, becasue the curves are most separated. The worst feature is Sepal Width, because of the high degree of over-lap.

Commentary: solutions and grading

I got lazy, and used Seaborn for the visualization. This is perfectly acceptable. The code from A1Q7 could be adapted here as well. A good job shows the graphs neatly. Colour and labelling is optional.

There are 2 lessons here.

1. If the class densities overlap a lot, classifiers can have low accuracy. This is not a flaw in Naive Bayes. This is a property of all classifiers.
2. If a classifier can look at combinations of features, there is still a chance that the classifer can be accurate. This is what the scatter plots tell us (above). So over-lapping features is not a guarantee of poor accuracy. This is a flaw in Naive Bayes. The way Naive Bayes combines the features may not give good accuracy.

Grading For full marks:

- The discussion should try to draw a connection between overlap in the density estimate, and the one-feature classifier accuracy.
- The discussion was short.

Part 4: Classifier Comparison

According to accuracy metric, the best one-feature classifier outperforms the four-feature classifier by a relatively small amount. This implies that the four-feature classifier is incorrectly classifying a few more test set examples.

Commentary: solutions and grading

Depending on the method used to calculate accuracy (simple split, or cross-validation), the answer can be different. For some simple splits, the four-feature classifier might be a bit better. This shows the robustness of cross-validation.

The discussion should reach the conclusion that one is a bit better. There's no need to explain the conclusion.

Grading For full marks

- You compared the best 1-feature classifier to the 4-feature classifier in terms of accuracy.
- Your discussion was short.

CMPT 423/820 Assignment 2 Question 2

Model Solution and Grading Scheme

Prologue: Importing and reading the Data ¶

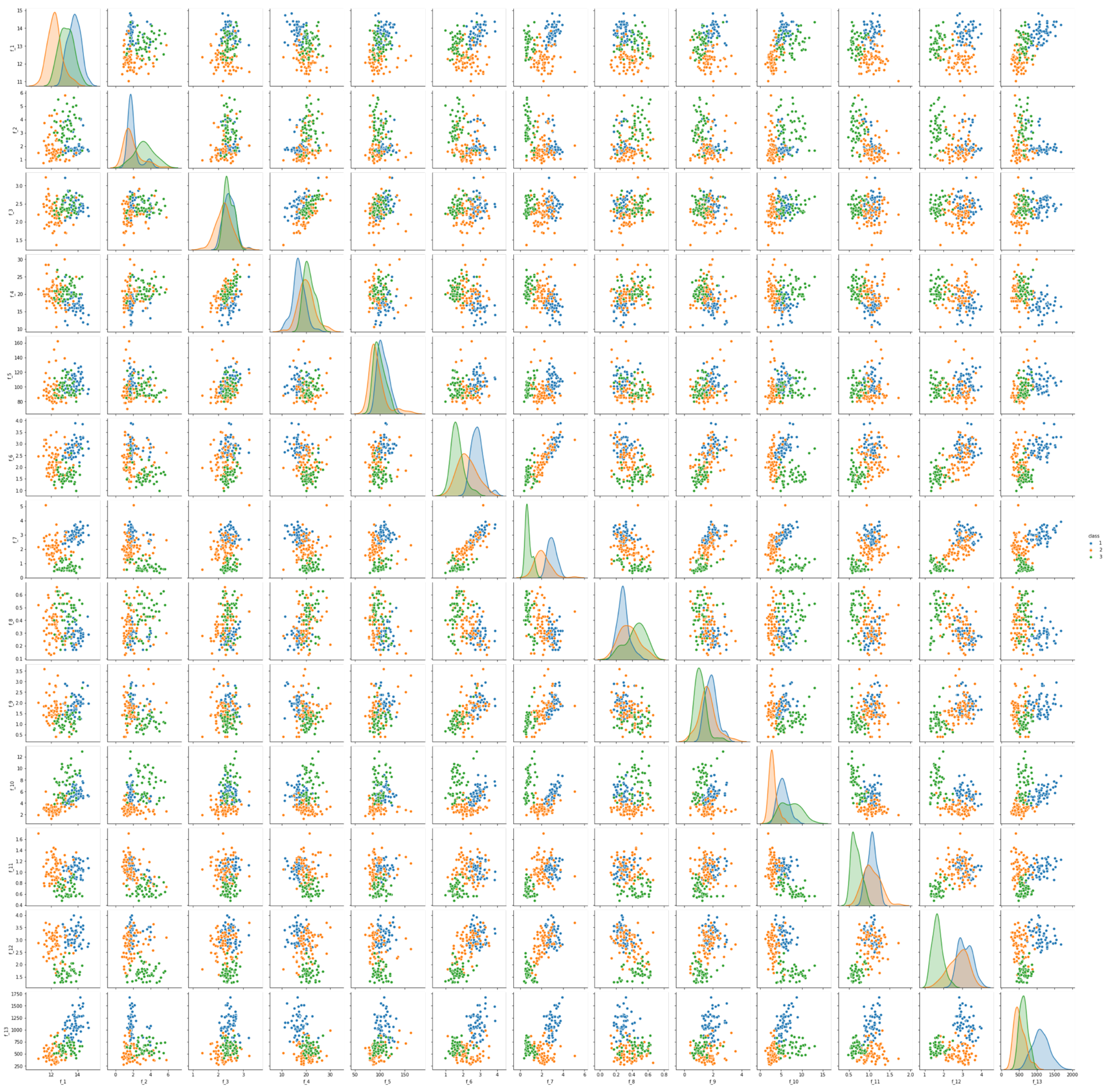
In [1]:
```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

# provide column names f_N
cnames = ['class']+["f_"+str(n) for n in range(1,14)]
dataframe = pd.read_csv('a2q3.csv',
                        header=None,
                        names=cnames,
                        index_col=False)
```

Density plots

The density plots are below.

In [2]:
```
sns.pairplot(dataframe, hue='class', diag_kind='kde')
plt.show()
```


f_1
f_2
f_3
f_4
f_5
f_6
f_7
f_8
f_9
f_10
f_11
f_12
f_13
class
1
2
3

Questions and Discussion

- The density plots show a lot of overlap between features for different classes. Features f_3, f_4, f_5 have a lot of overlap.
- Some of the features seem to have separations for two of the three classes, e.g., features f_1, f_6, f_7, f_{11}.
- Features f_{12}, f_{13} seem to separate one of the classes from the other two. In combination these two might be effective.

Which, if any, of the 13 features, would you pick as the single feature in a 1-feature classifier? Briefly explain your answer.

I'd choose feature f_7, because the density seems to have the least overlap for three classes. Still, there is some overlap, so a single-feature classifier based on this feature will still produce some errors.

Prior to building a classifier, do you think a classifier based on this data will have high accuracy? Briefly explain your answer.

Some of the features seem to have separations for two of the three classes, e.g., features f_1, f_6, f_7, f_{11}. I'd expect these features would have the highest one-feature classifier accuracy. Features f_{12}, f_{13} seem to separate one of the classes from the other two. In combination these two might be effective.

Scatter plots were not required by the question, but they do provide some information. The scatter plots suggest that the classes are clustered separately; i.e., there are regions where only one class is present. This does suggest that a different kind of classifier could do well. Looking at features f_{12} and f_{13}, a good classifier seems plausible.

Grading For full marks:

- Your density plots are correct, and neatly presented.
  - Scatter plots are not required!
  - Colour and labels are useful, but not required.
- You answers to the questions demonstrate you've assessed the features critically.
  - The best feature should not be one of the ones with substantial overlap.
  - The opinion about a classifier does not need to agree with my opinion for full marks. The opinion just needs to be supported in some way by the plots.

CMPT 423/820 Assignment 2 Question 3

Model Solution and Grading Scheme

Prologue: Importing and reading the Data

In [1]:
```python
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

from sklearn.naive_bayes import GaussianNB
from sklearn.neighbors import KNeighborsClassifier
from sklearn.tree import DecisionTreeClassifier

from sklearn import model_selection
from sklearn.preprocessing import StandardScaler

# give names to the columns
cnames = ['class']+["f_"+str(n) for n in range(1,14)]
dataframe = pd.read_csv('a2q3.csv',
                        header=None,
                        names=cnames,
                        index_col=False)

# Separate the features from the class
array = dataframe.values
X = array[:,1:14]
Y = array[:,0]

# Proportion of data as a part of validation set
validation_size = 0.20

# set up 10-fold cross validation
kfold = 10

# set up the performance metric once and for all
metric_name = 'f1_macro'
```

Part 1: K-Nearest Neighbours

K-Nearest Neighbours is a non-parametric classifier. In other words, it does not try to model the task environment by simplifying it. Instead, it remembers data, and uses a distance metric to determine if a new example is close to something it has already seen.

Scaling the data

Because all the data is numeric, a Euclidean distance metric is plausible (though not necessarily the only option). However, because distance depends on scale, we must scale the data to a standard range.

Scaling can be done using Python and Numpy, or scikit-learn modules. The code below uses a `StandardScaler` which scales each feature independently, so that the scaled feature has zero mean, and unit variance.

In [2]:
```
X_scaled = StandardScaler().fit_transform(X)

# cross validation to evaluate the performance
cv_results = model_selection.cross_val_score(KNeighborsClassifier(n_ne
ighbors=7),
                                             X_scaled, Y, cv=kfold, sc
oring=metric_name)

#displaying the mean and standard deviation of the prediction
msg = "%s: %f (%f)" % ('KNN performance with scaling', cv_results.mean
(), cv_results.std())
print(msg)
```

```
KNN performance with scaling: 0.966731 (0.036989)
```

KNN Comparison to unscaled data

To see the importance of scaling for KNN, observe the performance without scaling:

In [3]:
```
# cross validation to evaluate the performance
cv_results = model_selection.cross_val_score(KNeighborsClassifier(n_ne
ighbors=7),
                                             X, Y, cv=kfold, scoring=m
etric_name)

#displaying the mean and standard deviation of the prediction
msg = "%s: %f (%f)" % ('KNN performance without scaling', cv_results.m
ean(), cv_results.std())
print(msg)
```

```
KNN performance without scaling: 0.645687 (0.087908)
```

Clearly scaling improves the KNN classifier.

In [4]:
```
from sklearn.model_selection import train_test_split

# use the training set to tune the vlaue of min_samples_split
X_train, X_test, Y_train, Y_test = train_test_split(X_scaled, Y, test_
size=0.50)

for k in range(1,15,2):
    # cross validation to evaluate the performance
    cv_results = model_selection.cross_val_score(KNeighborsClassifier(
n_neighbors=k),
                                                 X_train, Y_train, cv=
kfold, scoring=metric_name)

    # displaying the mean and standard deviation of the prediction
    msg = "%s %d: %f (%f)" % ('KNN performance tuning k', k, cv_result
s.mean(), cv_results.std())
    print(msg)
```

```
KNN performance tuning k 1: 0.915873 (0.103934)
KNN performance tuning k 3: 0.940159 (0.078701)
KNN performance tuning k 5: 0.959153 (0.050271)
KNN performance tuning k 7: 0.959153 (0.050271)
KNN performance tuning k 9: 0.960000 (0.049267)
KNN performance tuning k 11: 0.958730 (0.071057)
KNN performance tuning k 13: 0.958730 (0.071057)
```

Now let's apply this choice of k to the validation set, and get an estimate of the performance on unseen data.

In [5]:
```
choice_k = 7

# cross validation to evaluate the performance
cv_results = model_selection.cross_val_score(KNeighborsClassifier(n_ne
ighbors=choice_k),
                                             X_test, Y_test, cv=kfold,
scoring=metric_name)

#displaying the mean and standard deviation of the prediction
msg = "%s: %f (%f)" % ('KNN performance on validation set after tuning
k', cv_results.mean(), cv_results.std())
print(msg)
```

```
KNN performance on validation set after tuning k: 0.988571 (0.034286
)
```

Now let's apply this choice of k to the whole data set, and get an estimate of the performance. We're doing this just to see the difference.

In [6]:
```
# cross validation to evaluate the performance
cv_results = model_selection.cross_val_score(KNeighborsClassifier(n_ne
ighbors=choice_k),
                                             X_scaled, Y, cv=kfold, sc
oring=metric_name)

#displaying the mean and standard deviation of the prediction
msg = "%s: %f (%f)" % ('KNN performance on whole data set after tuning
', cv_results.mean(), cv_results.std())
print(msg)
```

```
KNN performance on whole data set after tuning: 0.966731 (0.036989)
```

This is pretty close to the performance score on the validation set. It might be over-fitting a little, because we trained on half of that data.

Grading

For full marks:

- You fitted the KNN with a deliberately chosen k.
  - The best way to fit k is using a formal procedure like the above.
  - An informal attempt to find k by giving a few values a try is less good.
- You calculated the f_1 score on the data
  - A validation set is the best way
  - Using cross-validation with the whole data set is less good.

Part 2: Naive Bayes

Because Naive Bayes does not use a distance metric, we don't need to use the scaled data. Scaling should not affect the performance at all. Because Naive Bayes is so simple, there is no need to play around with settings and options in Scikit Learn. Below I use the validation sets just to keep the comparison fair.

In [7]:
```
# cross validation to evaluate the performance
cv_results = model_selection.cross_val_score(GaussianNB(),
                                              X_test, Y_test, cv=kfold,
scoring=metric_name)

#displaying the mean and standard deviation of the prediction
msg = "%s: %f (%f)" % ('NB performance on validation set', cv_results.
mean(), cv_results.std())
print(msg)
```

```
NB performance on validation set: 0.988571 (0.034286)
```

Now let's try the full data set.

In [8]:
```
# cross validation to evaluate the performance
cv_results = model_selection.cross_val_score(GaussianNB(),
                                             X, Y, cv=kfold, scoring=m
etric_name)

#displaying the mean and standard deviation of the prediction
msg = "%s: %f (%f)" % ('NB performance on full data set', cv_results.m
ean(), cv_results.std())
print(msg)
```

```
NB performance on full data set: 0.978306 (0.026591)
```

According to our measure, the so-called f_1 score (combining precision and recall), Naive Bayes is doing better than KNN.

Grading

For full marks:

- You fitted the Naive Bayes.
- You calculated the f_1 score on the data
  - A validation set is the best way
  - Using cross-validation with the whole data set is less good.

Part 3: Decision Trees

The decision tree classifier in Scikit Learn has a lot of options. You can choose:

- the purity measure, either `gini` or `entropy`
- the maximum depth of the tree
- the number of samples needed to allow a split
- lots of others!

Most of these options are used to prevent a Decision Tree from growing too big. Big trees tend to over-fit the training data, leading to poor generalization. We will see evidence of that here.

In the next cell, we use default options, to get a base line.

In [9]:
```
# cross validation to evaluate the performance
cv_results = model_selection.cross_val_score(DecisionTreeClassifier(),
                                             X, Y, cv=kfold, scoring=m
etric_name)

#displaying the mean and standard deviation of the prediction
msg = "%s: %f (%f)" % ('DT performance for default tree fitting', cv_r
esults.mean(), cv_results.std())
print(msg)
```

```
DT performance for default tree fitting: 0.849319 (0.135809)
```

That's worse than the other two.

I spent a good amount of time trying to find a good option to limit depth to avoid over-fitting. I tried most of the Scikit Learn options at least once, and I settled on trying to prevent splits when the number of samples was too low: `min_samples_split` .

In the cell below, I split into training and validation sets. I try various values for `min_samples_split` and display the results.

In [10]:
```
from sklearn.model_selection import train_test_split

# use the training set to tune the vlaue of min_samples_split
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.
50)

for dep in range(5,100,5):
    # cross validation to evaluate the performance
    cv_results = model_selection.cross_val_score(DecisionTreeClassifie
r(min_samples_split=dep),
                                                 X_train, Y_train, cv=
kfold, scoring=metric_name)

    # displaying the mean and standard deviation of the prediction
    msg = "%s %d: %f (%f)" % ('DT performance tuning min_samples_split
', dep, cv_results.mean(), cv_results.std())
    print(msg)
```

```
DT performance tuning min_samples_split 5: 0.862857 (0.081602)
DT performance tuning min_samples_split 10: 0.851429 (0.068555)
DT performance tuning min_samples_split 15: 0.838651 (0.088935)
DT performance tuning min_samples_split 20: 0.851429 (0.068555)
DT performance tuning min_samples_split 25: 0.808810 (0.142027)
DT performance tuning min_samples_split 30: 0.778016 (0.132841)
DT performance tuning min_samples_split 35: 0.733519 (0.112683)
DT performance tuning min_samples_split 40: 0.733519 (0.112683)
DT performance tuning min_samples_split 45: 0.733519 (0.112683)
DT performance tuning min_samples_split 50: 0.660539 (0.143192)
DT performance tuning min_samples_split 55: 0.598237 (0.132966)
DT performance tuning min_samples_split 60: 0.485625 (0.065475)
DT performance tuning min_samples_split 65: 0.485625 (0.065475)
DT performance tuning min_samples_split 70: 0.485625 (0.065475)
DT performance tuning min_samples_split 75: 0.485625 (0.065475)
DT performance tuning min_samples_split 80: 0.485625 (0.065475)
DT performance tuning min_samples_split 85: 0.183566 (0.018140)
DT performance tuning min_samples_split 90: 0.183566 (0.018140)
DT performance tuning min_samples_split 95: 0.183566 (0.018140)
```

It should be noted that the test-train split is randomized, and each random split is different. Decision tree classifiers are known to be sensitive to the data, which means that we have to expect that slightly different randomizations of the training set will result in possibly drastic differences in performance.

From the above tuning, it seems that setting the `min_samples_split=20` gives reliably best training set error. Now we have to see how that works on the validation set that was reserved for evaluation only:

In [11]:
```
# Use the best value for min_samples_split from previous cell
choice = 20

# cross validation to evaluate the performance
cv_results = model_selection.cross_val_score(DecisionTreeClassifier(mi
n_samples_split=choice),
                                              X_test, Y_test, cv=kfold,
scoring=metric_name)

#displaying the mean and standard deviation of the prediction
msg = "%s %d: %f (%f)" % ('DT performance using tuned value for min_sa
mples_split', choice, cv_results.mean(), cv_results.std())

print(msg)
```

```
DT performance using tuned value for min_samples_split 20: 0.891614
(0.115352)
```

The performance value obtained on the validation set is usually close to the value obtained on the training set while tuning, but the performance is not as stable as one might like.

Just for fun, let's see what happens when we apply this tuned value to the whole data set:

In [12]:
```
# cross validation to evaluate the performance
cv_results = model_selection.cross_val_score(DecisionTreeClassifier(mi
n_samples_split=choice),
                                              X, Y, cv=kfold, scoring=m
etric_name)

#displaying the mean and standard deviation of the prediction
msg = "%s %d: %f (%f)" % ('DT performance using tuned value for min_sa
mples_split', choice, cv_results.mean(), cv_results.std())

print(msg)
```

```
DT performance using tuned value for min_samples_split 20: 0.844258
(0.076784)
```

This is better than predicted by the validation set, but still not as good as either Naive Bayes or KNN. Decision trees are known to be sensitive to data. Randomization of the train-test split results in fairly wide variance in performance.

Grading

For full marks:

- You fitted the Decision Tree Classifier with a deliberately chosen options.
  - The best way to fit options is using a formal procedure like the above.
  - There's no need to fit all the possible options.
  - An informal attempt to find good values for the options by giving a few values a try is less good.
- You calculated the f_1 score on the data
  - A validation set is the best way
  - Using cross-validation with the whole data set is less good.

Discussion

Methodology The data was split into two equal halves, training and validation. Tuning options and settings was done using the training set, and 10-fold cross validation. Once the tuned settings and options were chosen, the method was applied to the validation set, and the performance evaluated using 10-fold cross validation.

Summarizing the results presented above, rounded to 3 significant figures (because more digits cloud the comparison without adding useful precision).

Method	Options	Validation	Full
K Nearest Neighbours	$k = 7$	0.960	0.967
Naive Bayes	(none)	0.956	0.978
Decision Tree	`min_samples_split` = 20	0.852	0.854

We can see that

1. KNN had the best performance on the validation set, but only by a little.
2. Naive Bayes had the best performance on the full dataset, but only by a little.
3. Decision trees had the worst performance on the validation set and the full dataset.

Looking at the difference between the performance on the validation set compared to the full data set, it seems that performance improves a little with the full data set. This makes sense in two ways. First, more data means there is less chance to find accidental patterns that causeclassification errors. Also, for KNN and DT, using the data with which we tuned the parameters means we're probably overfitting a little. On the other hand, the difference between these results is small, so the over-fitting is a minor issue.

Decision tree classification had trouble with this dataset because the separation between features was not easily represented in terms of vertical or horizontal splits in the feature space (see Question 2).

In terms of effort to produce these results:

1. Naive Bayes was the simplest.
2. Decision Trees was the most trouble.

If best possible classification performance is the criterion, then it's a toss up between KNN and NB. The difference is not appreciable unless there is a lot of money on the line with every error.

Naive Bayes models are small and fast: it's quick to fit, and quick to use for classification. KNN is trivial to fit, but troublesome to tune, and classification requires storage of the training dataset. No summarization of the dataset at assll. Decision trees are slow to fit, but fast to classify, but require a lot of tuning.

I choose Naive Bayes for this problem.

Grading

The main issue here is that there are so many different ways this question could go, depending on what methods and options were used. For example:

- There are 3 `f1` scoring options: `f1_micro` , `f1_macro` , and `f1_weighted` .
  - I had trouble with `f1_macro` ; its results were unpredictable and inscrutible.
- Using cross-validation or using simple test-train splits
- The size of the splits, and the number of folds.
  - I didn't notice a difference between 10-fold or 5-fold, but lots of variation with the test-train split at different proportions.
- Using `KFold()` for cross-validation vs `cv=10` in the cross-validation method.
  - using `cv=10` defaults **silently** to Strsatified Cross-validation, whereas `KFold()` does not.

For these reasons, the reported f_1 scores could be quite diffrent from what I have reported.
For full marks:

- Your Notebook is neat, and presents the problem and solution well
- You compared the three classifiers in terms of the f_1 score.
- Optional: You made other comparisons, e.g., ease of training, tuning, and computational performance issues.

CMPT 423/820 Assignment 2 Question 4

Model Solution and Grading Scheme

Currently, Scikit-Learn has 4 implementations of Naive Bayes. Each implementation assumes that all the features have the same kind of feature distribution. For example, the Scikit-Learn implementation of Gaussian Naive Bayes Classifier assumes that all features are numeric, and the histogram of the features given the class label are more-or-less bell shaped around a mean value. On the other hand, the Scikit-Learn implementation of the Categorical Naive Bayes Classifier assumes that all features are categorical. **This is a limitation of the software, not a theoretical limitation of Naive Bayes.**

In this question, you are invited to explain, or describe how we could use these two classifiers to handle mixed data.

Using Scikit Learn on mixed data

Suppose we have a categorical class label Y, some continuous features F and some categorical features C. For our purposes, it doesn't matter how many of each we have. I will derive a formula based on one of each, and generalize after that.

The classification formula for Naive Bayes doesn't care about the distinction between continuous or categorical fetures, so we have the same formula as before:

$$y(c, f) = \arg\max_Y P(Y|c, f)$$

As usual, we will start with the class conditional distribution, and apply Bayes Rule, and conditional independence:

$$P(Y|c, f) = \frac{P(c|Y)P(f|Y)P(Y)}{P(c, f)}$$

Unfortunately, Scikit Learn cannot fit this model with mixed data directly. Not yet, anyway. But It can fit continuous features using Gaussian Naive Bayes, and categorical features using Categorical Naive Bayes (in Version 0.22).

We can split the data into two parts, the categorical features, C (with Y), and the continuous features F (with Y). Fitting them independently gives us 2 classifiers:

$$y_1(c) = \arg\max_Y P(Y|c) = \arg\max_Y \frac{P(c|Y)P(Y)}{P(c)}$$
$$y_2(f) = \arg\max_Y P(Y|f) = \arg\max_Y \frac{P(f|Y)P(Y)}{P(f)}$$

The question is now, how to derive a formula for $y(c, f)$ given $y(c)$, and $y(f)$?

Since $y(c, f)$ uses $P(c|Y)$ and $P(f|Y)$, I will find expressions for these two factors. We start with

$$P(Y|c) = \frac{P(c|Y)P(Y)}{P(c)}$$

and rearrange to get

$$P(c|Y) = \frac{P(Y|c)P(c)}{P(Y)}$$

(this is also just Bayes Rule). Similarly:

$$P(f|Y) = \frac{P(Y|f)P(f)}{P(Y)}$$

Now we substitute into $y(c, f)$ and simplify as follows:

$$P(Y|c, f) = \frac{\frac{P(Y|c)P(c)}{P(Y)} \frac{P(Y|f)P(f)}{P(Y)} P(Y)}{P(c, f)} = \frac{P(Y|c)P(Y|f)}{P(Y)} \frac{P(c)P(f)}{P(c, f)}$$

I've gathered the factors that depend on Y and those that do not depend on Y. Note that $P(c)P(f) \neq P(c, f)$ in general.

Finally:

$$y(c, f) = \arg\max_Y P(Y|c, f) = \arg\max_Y \frac{P(Y|c)P(Y|f)}{P(Y)} \frac{P(c)P(f)}{P(c, f)} = \arg\max_Y \frac{P(Y|c)P(Y|f)}{P(Y)}$$

The last step is true because the factor involving c, f does not depend on Y, so it's constant, and cannot affect the maximization.

Scikit Learn allows us to access $P(X|Y)$ (where X is any feature), and $P(Y)$ for any Naive Bayes classifier it fits using:

- Any classifier
  - Method `predict_proba()` : this is $P(X|Y)$ where X is a feature.
  - Method `predict_log_proba()` : this is $\log P(X|Y)$ where X is a feature.
  - Attribute `class_prior_` : this is $P(Y)$
- Categorical NB
  - Attribute `feature_log_prob` : This is $\log P(X|Y)$ where X is a feature.
  - Attribute `class_log_prior` : This is $\log P(Y)$.
- Gaussian NB
  - Attribute `sigma_` , `theta_` : These are μ_{FY}, σ_{FY} for each feature F and each class Y. With these values, we can calculate $P(F|Y)$ using Numpy or Python's random modulem using the Normal distribution $\mathcal{N}(\mu_{FY}, \sigma^2_{FY})$

Now we ask the two fitted classifiers for all the $P(C|Y)$ and $P(F|Y)$. Then we can multiply these together (or better, add the log probabilities). Then we get $P(Y)$ (from either classifier -- they should report the same values), and divide (or subtract the log priors). Then it's a linear search through the probabilities, looking for the maximum.

Here's Pythonesque pseudo-code, ignoring details like multi-dimensional arrays returned by Scikit Learn:

```
# To fit the mixed classifier
clf_1 = CategoricalNB()
clf_2 = GaussianNB()
clf_1.fit(C,Y)
clf_1.fit(F,Y)

# to classify a given sample c,f:
lP_Y = clf_1.class_log_prior_
lP_C_Y = clf_1.feature_log_proba
mu, sigma = clf_2.mu_, clf_2.sigma
P_F_Y = NormalVariate(F, mu, sigma)

lP_Y_CF = lP_C_Y + lP_F_Y + log(P_F_Y) - lP_Y
select from lP_Y_CF the largest value
return the corresponding Y value
```

Grading Guideline

This question should be graded according to the level of accomplishment achieved. It is an open-ended question.

The derivation above is an example of a formal derivation, based on the probabilistic foundations of Naive Bayes.

- A good formal derivation (10/10) will:
  - Apply the two simpler formulae
  - Account for $P(Y)$ appearing in both of the smaller classifiers.
  - Rule out the constant factor as irrelevant.
  - Show that Scikit Learn can give us the information we need in the form of attributes or methods
- A weaker derivation (7/10) will
  - Multiply the results of the method `predict_proba()` which is $P(X|Y)$ together.
  - Ignore or omit $P(Y)$
  - Try to calculate the constant factor
  - Assume that the constant factor is equal to 1
- An adequate informal description (6/10) can:
  - suggest the use of `predict_proba()` for the question, without a proof or derivation
  - suggest asking both classifiers for a class, and use the answer as a vote; for this a tie breaking scheme needs to be included.
- A weak informal description (3/10) might:
  - suggest asking both classifiers for a class, and use the answer as a vote without a tie breaking scheme.
  - suggest some plausible approach not making use of the Naive Bayes classifiers provided by Scikit Learn.
- A totally inadequate description (0/10):
  - suggest an implausible approach

A demonstration requires:

- Creating a mixed data set, or finding a mixed dataset on line.
- Applying the technique above to the data set.
- Evaluating performance by:
  - Comparing accuracy (or error) of some examples
  - Evaluation of accuracy (or error) on a test set

Grading Summary

- Approach (10 possible marks)
  - Nothing submitted: 0/10
  - Totally inadequate: 0/10
  - Informal, but weak: 3/10
  - Adequate Informal: 6/10
  - Weak formal derivation: 7/10
  - Good formal derivation: 10/10
- Demonstration (5 possible marks)
  - Nothing submitted: 0/5
  - Showed a few examples: 2/5
  - Fitted with mixed training set and evaluated with mixed test set: 5/5