In [2]:
```
#########
#
#
# Date: 09 December 2017
# Author: David Ward
#
#
#########
```

The data chosen for this progam is the pre-processed data from the PRE folder. The data from the enron[i] subdirectories is moved to HAM and SPAM directories, respectively, in the PRE directory by the application. The datasets are then is split into *three* separate sets: Training, Test and Validation. The the validation set is first removed from the HAM and SPAM folders and then stored in a separate directory, named "validation", stored in the PRE directory.

The program's pre-processor cleans the data of all non-alphanumeric characters. The validation files are then read into PANDAS dataframes as subject and body for their respective classes (HAM or SPAM).

Similarly, the data that's left in the HAM and SPAM folders is then pre-processed to separate the subject from the body and clean the HAM and SPAM files of all non-alphanumeric characters. The data that is left in the HAM and SPAM files is then read into appropriate PANDAS dataframes (as EnronHAM or EnronSPAM) From there the HAM and SPAM dataframes are combined into a full Enron dataframe which is then split via the sklearn train_test_split module at a 70:30 ratio for Training and Test, respectively.

A series of classifiers tested under various parameters are examined, with each classifier's performance metrics plotted or graphed.

When the highest performing classifier is determined based upon the comparison of all classifiers tested, it is saved (pickled).

The pickled classifier is then loaded from disk and the validation data set is fed into the classifier. The classifier is then evaluated on similar metrics as before to determine it's prediction accuracy.

The libraries used are:

- Python built in libraries
- Scikit-Learn
- Pandas
- Numpy
- Seaborn

The following python modules are required.

In [648]:
```
%matplotlib inline
import pandas as pd
import numpy as np
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.naive_bayes import MultinomialNB
from sklearn import metrics
from sklearn.preprocessing import LabelEncoder
from sklearn.ensemble import ExtraTreesClassifier
from sklearn.preprocessing import LabelBinarizer
from sklearn.pipeline import Pipeline
from sklearn import model_selection
from sklearn.calibration import CalibratedClassifierCV
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.feature_extraction.text import HashingVectorizer
from sklearn.svm import SVC, NuSVC, LinearSVC
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.feature_selection import SelectKBest, f_classif
from sklearn.model_selection import GridSearchCV
import seaborn as sns
from sklearn.metrics import average_precision_score
import matplotlib.pyplot as plt
from sklearn.metrics import roc_auc_score
from sklearn.metrics import roc_curve, auc
import itertools
import shutil
from random import randint

from collections import Counter
from sklearn.feature_extraction import text
import re
import os
import pickle
```

The pre-processed version of the Enron dataset was chosen as the working dataset. **NOTE:** You *must* the sourceDir and destFolder variables towards the location of the Enron 'pre' directory for *BOTH* destFolder and sourceDir.

The duplicated values for both the destFolder and sourceDir were more for clarity during development and testing. The hamDir and spamDir are the locations for where our SPAM and HAM files will be stored after being copied from the enron[i] subdirectories within the "pre" folder.

In [649]:
```
# NOTE:
#
# sourceDIR and destFolder must be changed to your appropriate local directory
# where the PRE directory is located
#

sourceDir = "C:\\Users\\user\\Documents\\****\\Enron\\enron\\pre\\"
destFolder = "C:\\Users\\user\\Documents\*****\\Enron\\enron\\pre\\"

hamDir = destFolder + "HAM\\"
spamDir = destFolder + "SPAM\\"
```

List variables are created to store our subject, body (both SPAM and HAM) for our Training, Test and Validation sets.

In [650]:
```
subjectList = [] # Ham subjects
bodyList = [] #Ham Body
spamSubject = [] #Spam subject list
spamBody = [] # Spam Body List
validationSPAMBody = [] # Validation Set SPAM Body list
validationHAMBody = [] # Validation Set HAM Body list
validationSPAMSubject = [] # Validation Set SPAM Subject List
validationHAMSubject = [] # Validation Set HAM Subject List
```

getHAMFiles and **getSPAMFiles** read through all **enron[i]** subdirectories in the PRE directory for both HAM and SPAM mails. Both functions then move the files to their respective folders, HAM or SPAM.

In [651]:
```
def getHamFiles(sourceDir, destDir):
    for i in range (1,7):
        src_files = os.listdir(sourceDir +"enron" +  str(i) + "\\ham\\")
        for file in src_files:
            full_file_name = os.path.join((sourceDir + "enron" + str(i) + "\\ham\\"), file)
            if (os.path.isfile(full_file_name)):
                if not os.path.exists(destFolder + "HAM\\"):
                    os.makedirs(destFolder+"HAM\\")
                if (os.path.isfile(destFolder+"HAM\\"+file)):
                    pass
                else:
                    shutil.move(full_file_name, (destFolder+"HAM\\"))
```

In [652]:
```
def getSpamFiles(sourceDir, destDir):
    for i in range (1,7):
        src_files = os.listdir(sourceDir +"enron"  + str(i) + "\\spam\\")
        for file in src_files:
            full_file_name = os.path.join((sourceDir + "enron" +str(i) + "\\spam\\"), file)
            if (os.path.isfile(full_file_name)):
                if not os.path.exists(destFolder + "SPAM\\"):
                    os.makedirs(destFolder+"SPAM\\")
                if (os.path.isfile(destFolder+"SPAM\\"+file)):
                    pass
                else:
                    shutil.move(full_file_name, (destFolder+"SPAM\\"))
```

We call both functions and supply the appropriate directories.

In [653]:
```
getHamFiles(sourceDir,destFolder )
getSpamFiles(sourceDir, destFolder)
```

In order to perform a validation test on our final classifier, we must select a sufficient number of SPAM and HAM mails for our validation data set.

getRangeFiles generates a 250 item array of random integers between 0 and 16500. An array (in this instance 250) of HAM and SPAM emails will be selected based upon their index number. The index number will be between 0 and 16500.

In [654]:
```
def getRangeFiles(fRange):
    fileIndex = []
    for i in range (0, fRange):
        if (randint(0,16500)) not in fileIndex:
            fileIndex.append(randint(0,16500))
    return ((fileIndex))
```

makeValidation iterates through the specified HAM or SPAM directory and moves files with indexes specified in fileIndex to their respective subdirectories in the validation directory.

```
In [655]: def makeValidation(fType):
    src_files = os.listdir(destFolder + fType + "\\")
    if not os.path.exists(destFolder + fType + "\\"):
        os.makedirs(destFolder + fType + "\\")
    for i in getRangeFiles(250):
        full_file_name = os.path.join((destFolder + fType + "\\"), src_files[i])
        if (os.path.isfile(full_file_name)):
            if not os.path.exists(destFolder + "validation" + "\\" + fType + "\\"):
                os.makedirs(destFolder + "validation" +"\\" +  fType + "\\")
            shutil.move(full_file_name, (destFolder+"validation" +"\\" + fType +"\\"))
```

getValidationMails iterates over the specified class directory and does the following in order:

1. Reads the file
2. Strips the file of all special characters not defined.
3. Writes the modified string back to the file
4. Copies the subject from the mail and appends it to the HAM/SPAM validation subject array
5. Copies the body from the mail and appends it to the HAM/SPAM validation body array.

```
In [656]: def getValidationMails(fType):

    validDir = (destFolder + "\\validation" + "\\" + fType + "\\")
    for subdir, dirs, files in os.walk(validDir):
        for file in files:
            string = open(validDir+file).read()
            new_str = re.sub('[^a-zA-Z0-9\n\:]', ' ', string)
            open(validDir+file, 'w').write(new_str)

    for files in os.walk(validDir):
        for file in files[2]:
            with open (validDir+file, 'r') as f:
                first_line = f.readline().rstrip()
                if fType == "HAM":
                    validationHAMSubject.append(first_line)
                if fType == "SPAM":
                    validationSPAMSubject.append(first_line)
                body_line = f.read().split('\n')
                if body_line == '':
                    pass
                if body_line == ' ':
                    pass
                if body_line == '[]':
                    pass
                if fType == "HAM":
                        validationHAMBody.append(body_line)
                if fType == "SPAM":
                        validationSPAMBody.append(body_line)
```

We then call the functions and provide the appropriate classes.

```
In [657]: makeValidation("HAM")
makeValidation("SPAM")
getValidationMails("HAM")
getValidationMails("SPAM")
```

Similarly with **getValidationMails**, we do the same for HAM and SPAM to clean the mails and populate the appropriate arrays.

In [658]:
```python
# Prepare ham files
for subdir, dirs, files in os.walk(hamDir):

    for file in files:

        string = open(hamDir+file).read()
        new_str = re.sub('[^a-zA-Z0-9\n\:]', ' ', string)
        open(hamDir+file, 'w').write(new_str)


for files in os.walk(hamDir):
    for file in files[2]:
        with open (hamDir+file, 'r') as f:
            first_line = f.readline().rstrip()
            subjectList.append(first_line)
            body_line = f.read().split('\n')
            if body_line == '':
                pass
            if body_line == ' ':
                pass
            if body_line == '[]':
                pass
            else:
                bodyList.append(body_line)
```

In [659]:
```python
# Prepare SPAM files

for subdir, dirs, files in os.walk(spamDir):
    for file in files:
        string = open(spamDir+file, encoding='latin-1').read()

        new_str = re.sub('[^a-zA-Z0-9\n\:]', ' ', string)

        open(spamDir+file, 'w').write(new_str)

for files in os.walk(spamDir):
        for file in files[2]:
            with open (spamDir+file, 'r', encoding='latin-1') as f:
                first_line = f.readline().rstrip()
                spamSubject.append(first_line)
                body_line = f.read().split('\n')
                if body_line == '':
                    pass
                if body_line == ' ':
                    pass
                if body_line == '[]':
                    pass
                else:
                    spamBody.append(body_line)
```

Next, dataframes must be created from the data stored in the arrays.

In [660]:
```python
HAM = 'HAM'
SPAM = 'SPAM'
```

In [661]:
```python
enronHAM = pd.DataFrame({'Subject': subjectList, 'Body': bodyList, 'Classification': HAM})
enronSPAM = pd.DataFrame({'Subject': spamSubject, 'Body': spamBody, 'Classification': SPAM})
enronValidationSPAM = pd.DataFrame({'Subject': validationSPAMSubject, 'Body': validationSPAMBody, 'Classification': SP
AM})
enronValidationHAM = pd.DataFrame({'Subject': validationHAMSubject, 'Body': validationHAMBody, 'Classification': HAM})
```

The newly created DataFrames are then cleaned by removing any rows with contain a null value in the Body column.

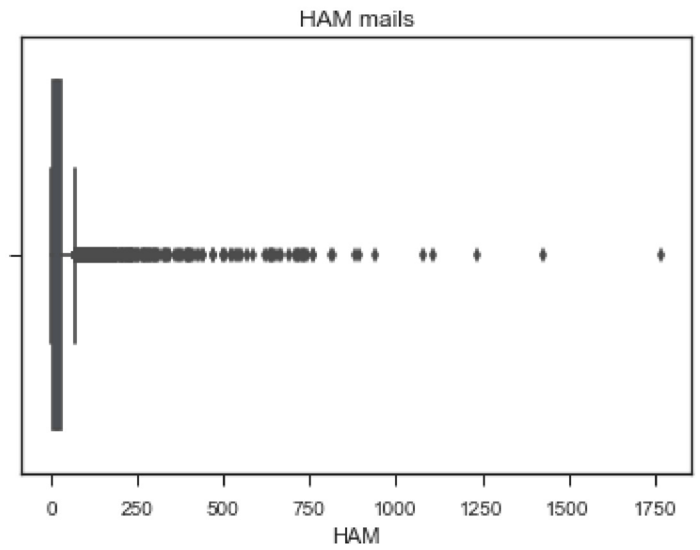
In [662]:
```python
enronSPAM['Body'] = enronSPAM['Body'].dropna(how='any')
enronHAM['Body'] = enronHAM['Body'].dropna(how='any')
enronValidationSPAM['Body'] = enronValidationSPAM['Body'].dropna(how='any')
enronValidationHAM['Body'] = enronValidationHAM['Body'].dropna(how='any')
```

Some initial statistics on the data is then performed. Here, the sizes of our datasets (now minus the validation set) is measured with figures and boxplots.

In [663]:
```
# HAM statistics

print ("Mail sizes")
bodyLengthHAM = []
for mails in (enronHAM.Body.str.len()):
    bodyLengthHAM.append(mails)

bodyLengthSPAM = []
for mails in (enronSPAM.Body.str.len()):
    bodyLengthSPAM.append(mails)

hamMailSizes = pd.DataFrame({'HAM':bodyLengthHAM})


hamMailSizes.HAM = hamMailSizes.HAM.astype(int)
print ("Total HAM mail size:", hamMailSizes.HAM.count())
print ("Mean HAM mail size:", hamMailSizes.HAM.mean())

sns.boxplot(hamMailSizes.HAM).set_title("HAM mails")
```

```
Mail sizes
Total HAM mail size: 16301
Mean HAM mail size: 25.59168149193301
```

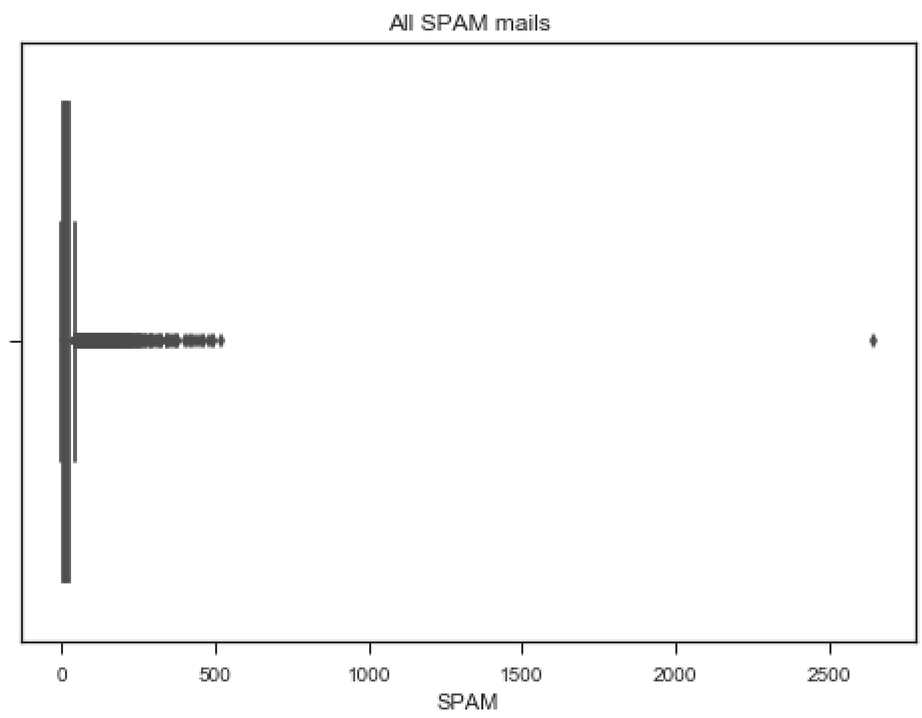
Out[663]: `<matplotlib.text.Text at 0xcd42860>`



In [664]:
```
# Spam Mail statistics
bodyLengthSPAM = []
for mails in (enronSPAM.Body.str.len()):
    bodyLengthSPAM.append(mails)
spamMailSizes = pd.DataFrame({'SPAM':bodyLengthSPAM})
print ("Mean spam mail size:", spamMailSizes.SPAM.mean())
print ("Total Spam Mails: ", spamMailSizes.SPAM.count())

spamMailSizes.SPAM = spamMailSizes.SPAM.astype(int)
sns.set(style='ticks')
sns.boxplot(spamMailSizes.SPAM).set_title("All SPAM mails")
```

```
Mean spam mail size: 22.77244234180958
Total Spam Mails:  16910
```

Out[664]: `<matplotlib.text.Text at 0x54555f98>`



Both the SPAM and HAM datasets now need to be combined to become the full Enron dataset. This will allow us to split the data into training and test sets later on. The validation sets are also combined to build the separate validation set for use on our final chosen model.

```
In [665]: combinedEnron = [enronSPAM, enronHAM]

fullEnron = pd.concat(combinedEnron)

validationCombined = [enronValidationHAM, enronValidationSPAM]

validationSet = pd.concat(validationCombined)
```

To prevent bias in our training and test sets and to ensure the data is distributed sufficiently, the dataframes must be reindexed so the data can be split.

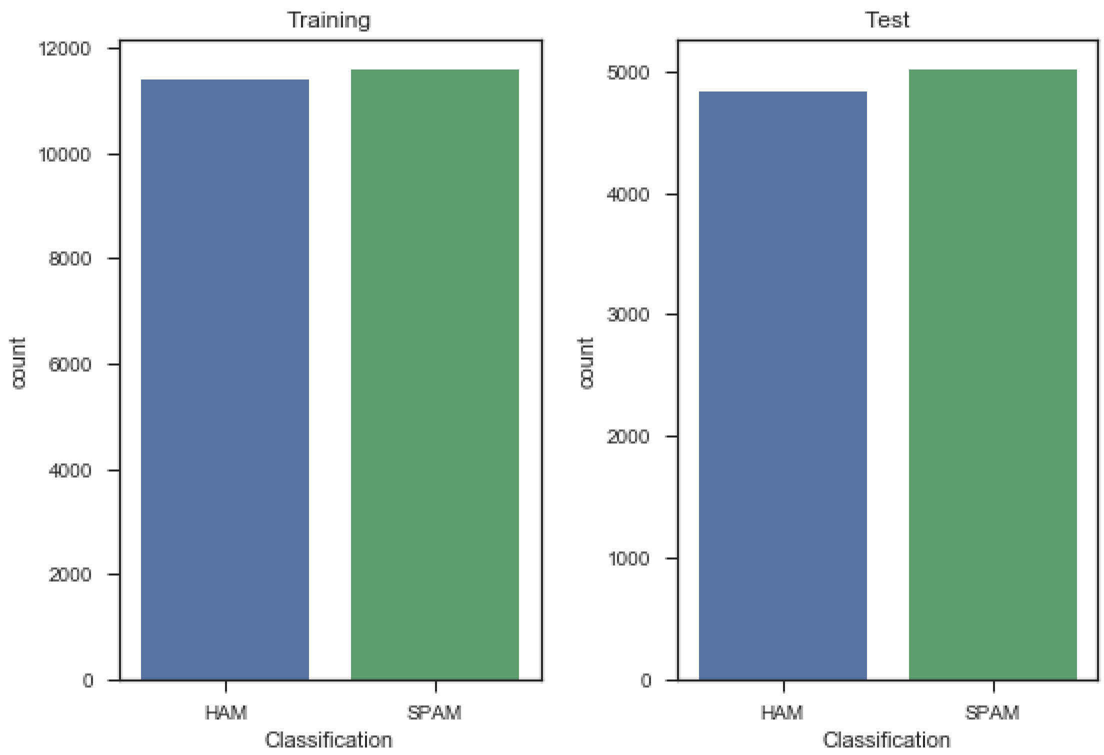
```
In [666]: #rebuild the index
fullEnron = fullEnron.reset_index(drop=True)
fullEnron['Body'] = fullEnron.Body.apply(','.join)
fullEnron = fullEnron.ix[fullEnron['Body'] != ""]
fullEnron = fullEnron.reindex(np.random.permutation(fullEnron.index))

fullEnron['Body'] = fullEnron['Body'].str.replace(',', ' ')

# Do the same for the validation set
validationSet = validationSet.reset_index(drop=True)
validationSet['Body'] = validationSet.Body.apply(','.join)
validationSet = validationSet.ix[validationSet['Body'] != ""]
validationSet = validationSet.reindex(np.random.permutation(validationSet.index))
```

We now split the data 70:30 for our training and test sets, respectively.

```
In [667]: # Create training set
trainingEnron, testEnron = train_test_split(fullEnron, test_size=0.3)
```

The data is now split. The training and test splits can now be analysed.

```
In [668]: # Training and Test set stats

fig, ax = plt.subplots(1,2)
sns.countplot(trainingEnron.Classification, ax=ax[0])
sns.countplot(testEnron.Classification, ax=ax[1])

ax[0].set_title("Training")
ax[1].set_title("Test")
fig.tight_layout()
```



Statistics for total mails, both HAM and SPAM in our Training data set:

```
In [669]: # Training dataset
print ("Total training mails: ", trainingEnron.Body.count())
trainBodyLength = []
for mails in (trainingEnron.Body.str.len()):
    trainBodyLength.append(mails)
trainingMailSizes = pd.DataFrame({'Body':trainBodyLength})

print ("Mean training mail size: ", trainingMailSizes.Body.mean())
```

```
Total training mails:  22993
Mean training mail size:  1505.3376679859089
```

Statistics for total mails, both HAM and SPAM in our Test data set:

In [670]:
```
# Test Dataset
print ("Total testing mails: ", testEnron.Body.count())
testBodyLength = []
for mails in (testEnron.Body.str.len()):
    testBodyLength.append(mails)
testMailSizes = pd.DataFrame({'Body':testBodyLength})

print ("Mean test mail size: ", testMailSizes.Body.mean())
```

```
Total testing mails:  9855
Mean test mail size:  1445.5777777777778
```

A count of all the words in our training data set is required. With this count, the most common / frequent words are then calculated. The frequency of words may be useful in classifying the documents as either SPAM or HAM.

In [671]:
```
# Count our words in our training set
results = Counter()
countV = CountVectorizer(stop_words='english')
countV.fit_transform(trainingEnron.Body)
trainingEnron.Body.str.lower().str.split().apply(results.update)
```

Out[671]:
```
27389    None
13571    None
8778     None
14027    None
23224    None
27650    None
24431    None
4230     None
25967    None
18923    None
18009    None
16794    None
1996     None
27875    None
1202     None
22586    None
11472    None
31156    None
30167    None
11896    None
22545    None
5943     None
24390    None
17828    None
8877     None
16696    None
32392    None
8192     None
2372     None
18355    None
         ...
23780    None
3196     None
8882     None
23712    None
16665    None
9384     None
24359    None
19333    None
18457    None
30763    None
9427     None
8942     None
3469     None
6543     None
21426    None
33088    None
21329    None
21316    None
5803     None
13162    None
22740    None
17210    None
11286    None
24558    None
20330    None
24687    None
30388    None
21782    None
21451    None
28912    None
Name: Body, dtype: object
```

Return the top 20 words.

In [672]:
```
topTwenty = (results.most_common(20))
print (topTwenty)
```

```
[('the', 202889), ('to', 147228), (':', 110581), ('and', 108469), ('of', 103029), ('a', 81692), ('in', 74381), ('fo
r', 56400), ('you', 55402), ('is', 49325), ('this', 43803), ('enron', 41638), ('on', 40617), ('that', 39627), ('i', 3
9072), ('s', 36001), ('be', 32663), ('with', 32627), ('your', 31416), ('we', 29831)]
```

The top 20 words seem to be similar to those found in a stopwords file. Only one word, "Enron" is a standout. The word count required is then increased to 50, and all words are included in the stop_words file.

In [673]:
```
# Examine the top 50, which most seem to be stop words. Append top 50 to stop_words.
dictAr = (results.most_common(50))
new_words = []
for key, value in dictAr:
    new_words.append(key)

stop_words = text.ENGLISH_STOP_WORDS.union(new_words)
```

The following classifiers have been tested.

1. Multinomial Naive Bayes - with CountVectorizer and a MultiNomial Naive Bayes Classifier.
2. MultiNomial Naive Bayes - with TfidfVectorizer and a Multinomial Naive Bayes Classifer.
3. Support Vector Classifier - with TfidfVectorizer and a LinearSVC classifier.
4. Support Vector Classifier - with CountVectorizer and a LinearSVC classifier.
5. Support Vector Classifier - with HashingVectorizer and a LinearSVC classifier.
6. Linear Regression Classifier - with TfidfVectorizer and a LinearRegression classifier.

Variations of these classifiers have been implemented, with stop word removal and document frequency variations.

In [674]:
```
# Multinomial Naive Bayes Pipeline with stop word removal

MNBpipeline = Pipeline([
    ('vectorizer',  CountVectorizer(stop_words=stop_words)),
    ('classifier',  MultinomialNB()) ])

MNBpipeline.fit(trainingEnron.Body.values, trainingEnron.Classification.values)

MNBPipelineClass =  (MNBpipeline.predict(testEnron.Body.values))
print ("Multinomial BP: ", metrics.accuracy_score(testEnron.Classification.values, MNBPipelineClass))
```

```
Multinomial BP:  0.98437341451
```

In [675]:
```
# MNB with TFIDF with sublinear and max document frequency

MNBTFpipeline = Pipeline([
    ('vectorizer',  TfidfVectorizer( sublinear_tf=True, max_df=0.69
                                 )),
    ('classifier',  MultinomialNB()) ])

MNBTFpipeline.fit(trainingEnron.Body.values, trainingEnron.Classification.values)
MNBTFPipelineClass =  (MNBTFpipeline.predict(testEnron.Body.values))
MNFIT = MNBTFpipeline.fit(trainingEnron.Body.values, trainingEnron.Classification.values)

print ("Multinomial BP with TF: ", metrics.accuracy_score(testEnron.Classification.values, MNBTFPipelineClass))
```

```
Multinomial BP with TF:  0.983358701167
```

In [676]:
```
# SVM with TFIDF and Calibrated LinearSVC

linSVCC = CalibratedClassifierCV(LinearSVC())
LinearSVCpipeline = Pipeline([
    ('vectorizer',  TfidfVectorizer(sublinear_tf=True, max_df=0.69, stop_words=stop_words)),
    ('classifier',  linSVCC) ])

LinearSVCpipeline.fit(trainingEnron.Body.values, trainingEnron.Classification.values)
LinearSVCpredictClass =  (LinearSVCpipeline.predict(testEnron.Body.values))
LinearSVCprob = LinearSVCpipeline.fit(trainingEnron.Body.values, trainingEnron.Classification.values)

LinearSVCFit = (LinearSVCpipeline.fit(trainingEnron.Body.values, trainingEnron.Classification.values))
print ("LinearSVC Score: ", metrics.accuracy_score(testEnron.Classification.values, LinearSVCpredictClass))
```

```
LinearSVC Score:  0.99066463724
```

In [677]:
```
# SVM Classifier with Linear Classification and hinge loss

SVCCVpipeline = Pipeline([
    ('vectorizer',  CountVectorizer()),
    ('classifier',  LinearSVC(loss='hinge')) ])

SVCCVpipeline.fit(trainingEnron.Body.values, trainingEnron.Classification.values)
SVCCVpredictClass =  (SVCCVpipeline.predict(testEnron.Body.values))
print ("SVCCV Score: ", metrics.accuracy_score(testEnron.Classification.values, SVCCVpredictClass))
```

```
SVCCV Score:  0.98061897514
```

In [678]:
```
# SVM with HashingVectorizer and LinearSVC
SVHVpipeline = Pipeline([
        ('vectorizer', HashingVectorizer()),
        ('classifier', LinearSVC()) ])



SVHVpipeline.fit(trainingEnron.Body.values, trainingEnron.Classification.values)

SVHVpredictClass = SVHVpipeline.predict(testEnron.Body.values)
print ("SVHV Score: ", metrics.accuracy_score(testEnron.Classification.values, SVHVpredictClass))
```

```
SVHV Score:  0.987519025875
```

In [679]:
```
# SVM Pipeline with CountVectorizer, stop word removal, LinearSVC and hinge loss.

SVMpipeline = Pipeline([
        ('vectorizer', CountVectorizer(stop_words=stop_words)),
        ('classifier', LinearSVC(loss='hinge')) ])



SVMpipeline.fit(trainingEnron.Body.values, trainingEnron.Classification.values)

SVMpredictClass = SVMpipeline.predict(testEnron.Body.values)
print ("SVM Score: ", metrics.accuracy_score(testEnron.Classification.values, SVMpredictClass))
```

```
SVM Score:  0.978893962456
```

In [680]:
```
# Logistic Regression with TFIDFvectorizer

LRpipeline = Pipeline([
        ('vectorizer', TfidfVectorizer()),
        ('classifier', LogisticRegression())
    ])


LRpipeline.fit(trainingEnron.Body.values, trainingEnron.Classification.values)

LRpredClass = LRpipeline.predict(testEnron.Body.values)
print ("Linear Regression: ", metrics.accuracy_score(testEnron.Classification.values, LRpredClass))
```

```
Linear Regression:  0.981735159817
```

By the classifier results, the SVM with TFIDF and a Calibrated LinearSVC with stop word removal and document frequency is the clear winner. The classifiers are evaluated again with a confusion matrix.

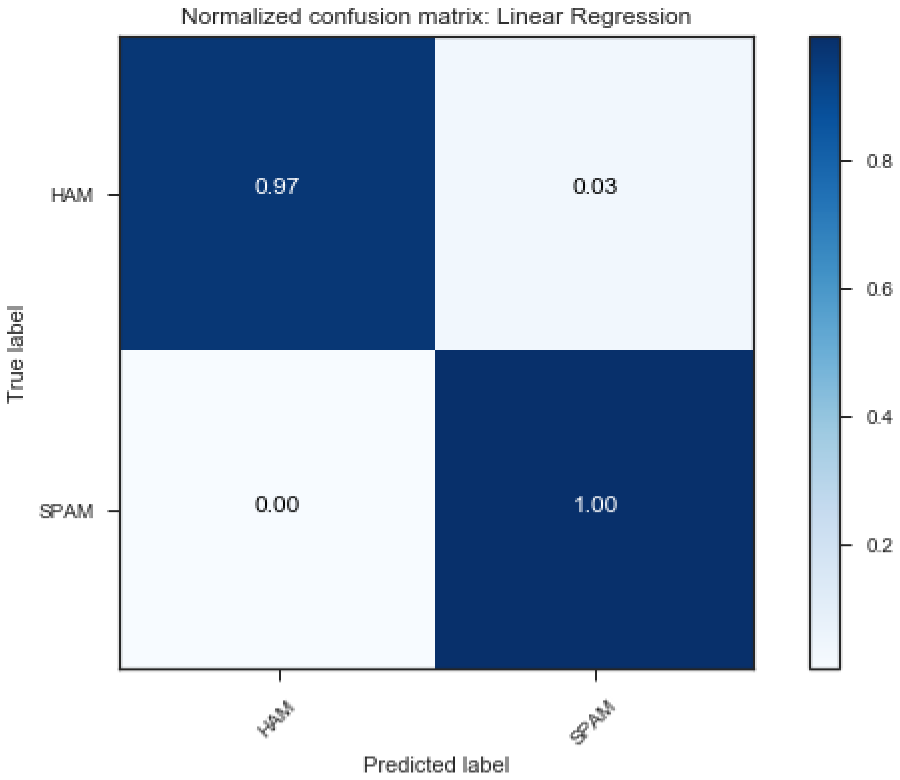
In [681]:
```
## Code modified from example given on SKLEARN documentation

def plot_confusion_matrix(cm, classes,
                          normalize=False,
                          title='Confusion matrix',
                          cmap=plt.cm.Blues):
    """
    This function prints and plots the confusion matrix.
    Normalization can be applied by setting `normalize=True`.
    """
    if normalize:
        cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
        print("Normalized confusion matrix")
    else:
        print('Confusion matrix, without normalization')

    print(cm)

    plt.imshow(cm, interpolation='nearest', cmap=cmap)
    plt.title(title)
    plt.colorbar()
    tick_marks = np.arange(len(classes))
    plt.xticks(tick_marks, classes, rotation=45)
    plt.yticks(tick_marks, classes)

    fmt = '.2f' if normalize else 'd'
    thresh = cm.max() / 2.
    for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
        plt.text(j, i, format(cm[i, j], fmt),
                 horizontalalignment="center",
                 color="white" if cm[i, j] > thresh else "black")

    plt.tight_layout()
    plt.ylabel('True label')
    plt.xlabel('Predicted label')
```

In [682]:
```
class_names = ['HAM', 'SPAM']
```

The classifiers are now evaluated with a confusion matrix.

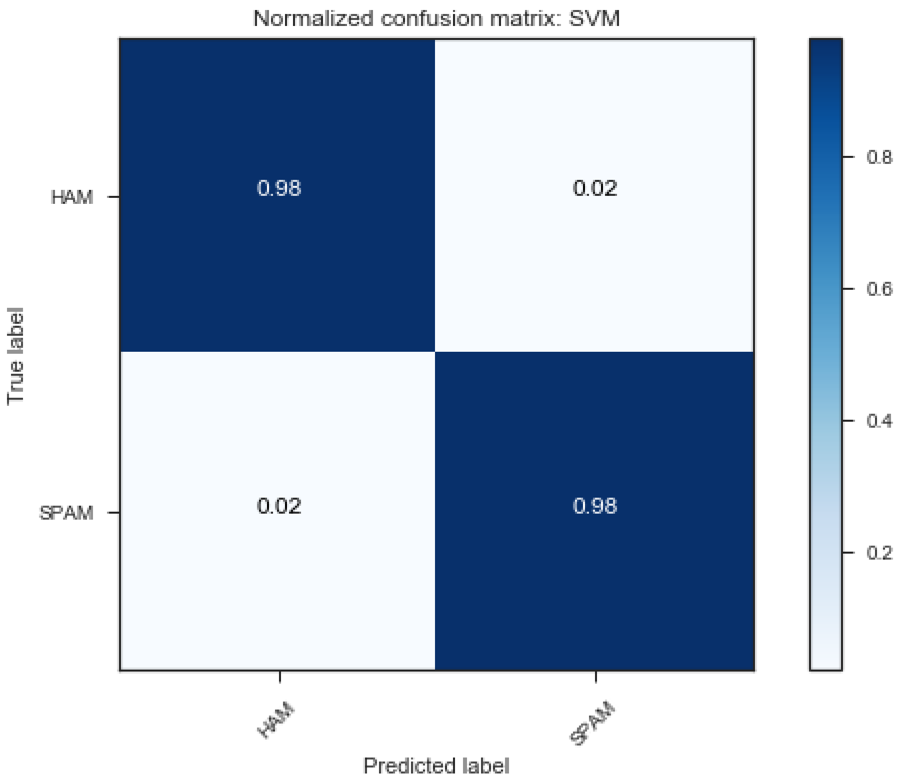
In [683]:
```
# Linear Regression plot
lrMetrics = (metrics.confusion_matrix(testEnron.Classification.values, LRpredClass))
plt.figure()
plot_confusion_matrix(lrMetrics, classes=class_names, normalize=True,
                      title='Normalized confusion matrix: Linear Regression')
```

```
Normalized confusion matrix
[[ 0.96735537  0.03264463]
 [ 0.00438684  0.99561316]]
```



In [684]:
```
# SVM Pipeline with CountVectorizer, stop word removal, LinearSVC and hinge loss.

svmMetrics = (metrics.confusion_matrix(testEnron.Classification.values, SVMpredictClass))
plt.figure()
plot_confusion_matrix(svmMetrics, classes=class_names, normalize=True, title='Normalized confusion matrix: SVM')
```

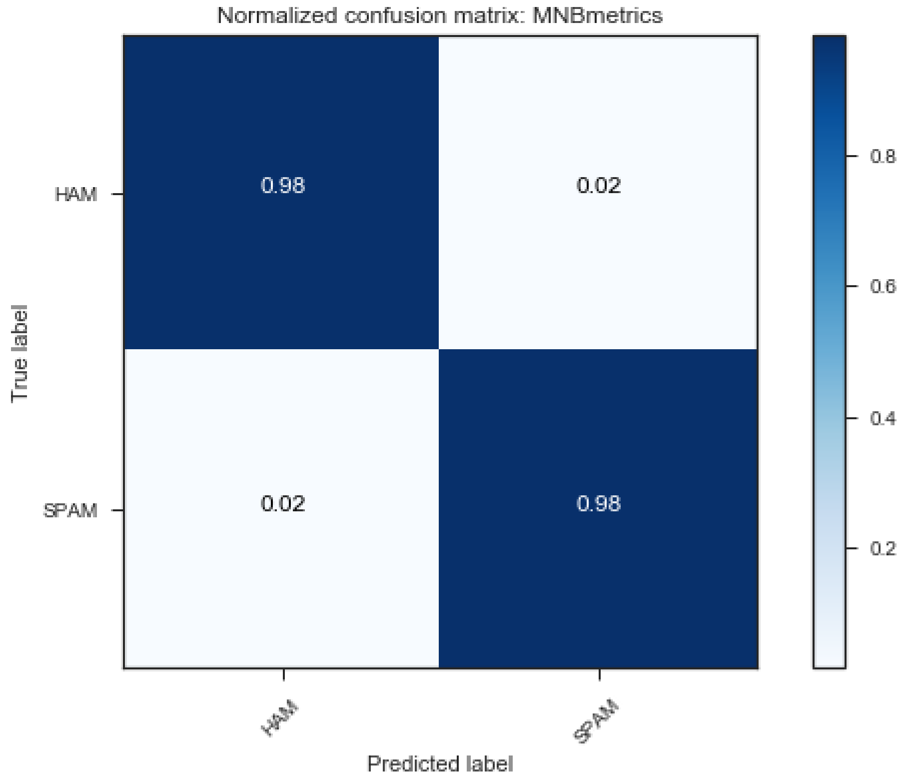
```
Normalized confusion matrix
[[ 0.97830579  0.02169421]
 [ 0.02053838  0.97946162]]
```



In [685]:

```
# Multinomial Naive Bayes Pipeline with stop word removal

MNBmetrics = (metrics.confusion_matrix(testEnron.Classification.values, MNBPipelineClass))
plt.figure()

plot_confusion_matrix(MNBmetrics, classes=class_names, normalize=True, title='Normalized confusion matrix: MNBmetrics'
)
```

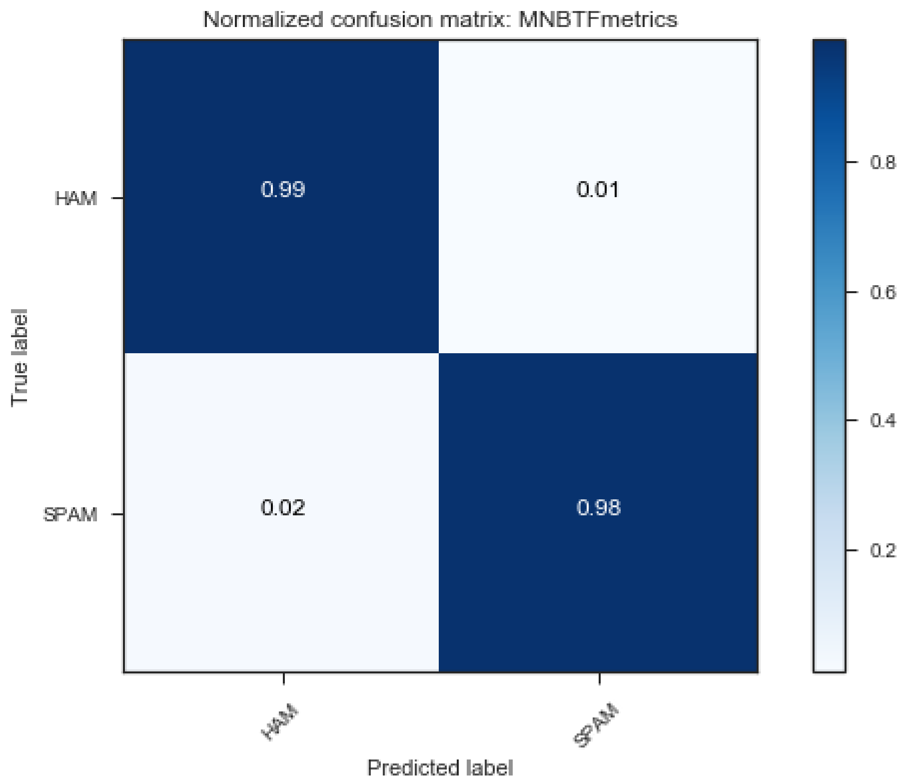
```
Normalized confusion matrix
[[ 0.98471074  0.01528926]
 [ 0.01595214  0.98404786]]
```



In [686]:

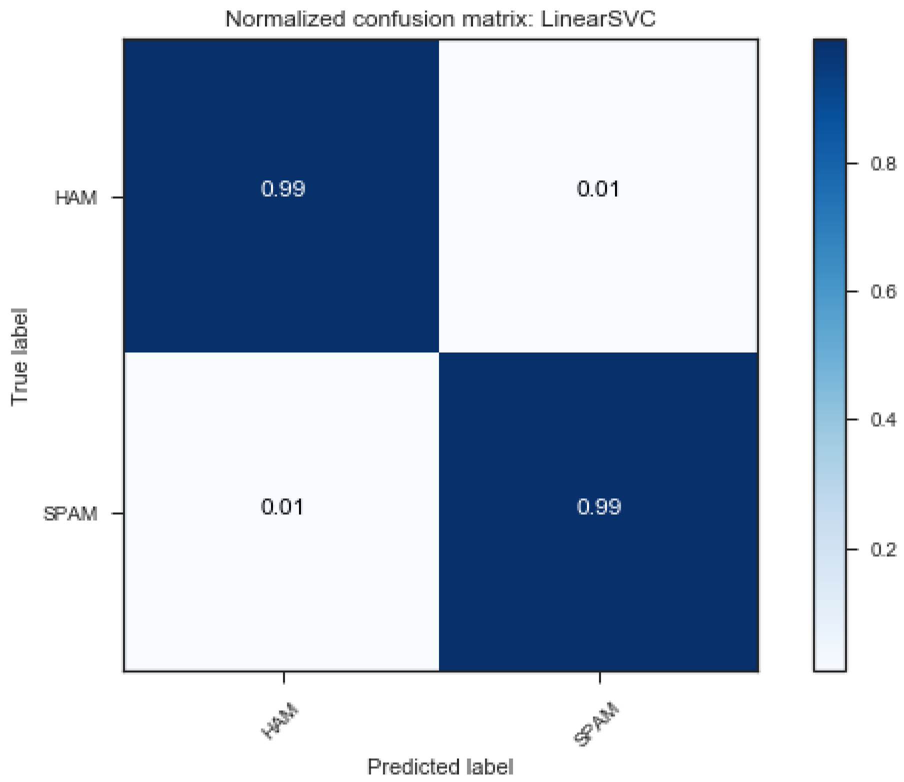
```
# MNB with TFIDF with sublinear and max document frequency

MNBTFmetrics = (metrics.confusion_matrix(testEnron.Classification.values, MNBTFPipelineClass))
plt.figure()

plot_confusion_matrix(MNBTFmetrics, classes=class_names, normalize=True, title='Normalized confusion matrix: MNBTFmetr
ics')
```

```
Normalized confusion matrix
[[ 0.98904959  0.01095041]
 [ 0.0221336   0.9778664 ]]
```



In [687]:
```
# Linear SVC TF Plot
LinearSVCMetrics = (metrics.confusion_matrix(testEnron.Classification.values, LinearSVCpredictClass))
plt.figure()
plot_confusion_matrix(LinearSVCMetrics, classes=class_names, normalize=True, title='Normalized confusion matrix: Linea
rSVC')
```

```
Normalized confusion matrix
[[ 0.98884298  0.01115702]
 [ 0.00757727  0.99242273]]
```



The confusion matrix confirms what the classifier metrics stated previously;the SVM with TFIDF and a Calibrated LinearSVC with stop word removal and document frequency is the most accurate of all the other classifiers trained.

Statistics on the training data and the results of our training data can be observed to show the amount of data trained on and the overall accuracy of our winning classifer.

In [688]:
```
# The counts of SPAM and HAM emails in our training set

print ("SPAM / HAM", trainingEnron.Classification.value_counts())
```

```
SPAM / HAM SPAM    11583
HAM     11410
Name: Classification, dtype: int64
```

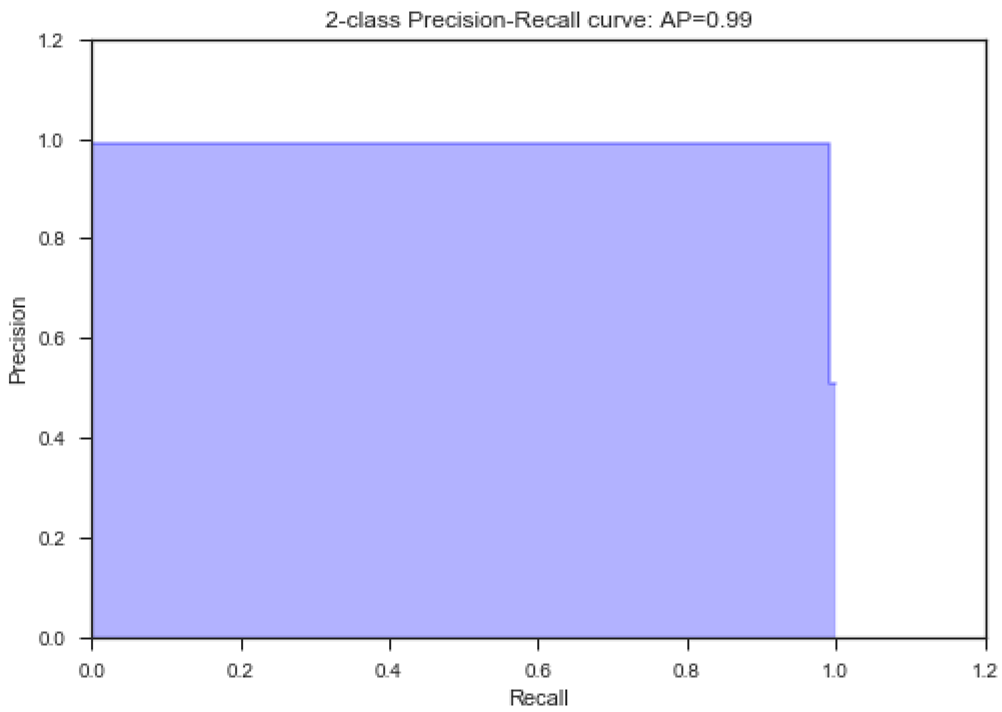
In [689]:
```
# Classification Report for our best classifier

print (metrics.classification_report(testEnron.Classification.values, LinearSVCpredictClass ))
print ("mean classification accuracy")
print (np.mean(testEnron.Classification.values == LinearSVCpredictClass))
```

```
             precision    recall  f1-score   support

        HAM       0.99      0.99      0.99      4840
       SPAM       0.99      0.99      0.99      5015

avg / total       0.99      0.99      0.99      9855

mean classification accuracy
0.99066463724
```

The precision recall curve for our most accurate classifer

In [690]:
```
## Precision Recall Curve
lb = LabelBinarizer()


trueValues = lb.fit_transform(testEnron.Classification.values)
LabelBinarizer(neg_label=0, pos_label=1, sparse_output=False)
predValues = lb.fit_transform(LinearSVCpredictClass)
precision, recall, _ = metrics.precision_recall_curve(trueValues, predValues)
average_precision = average_precision_score(trueValues, predValues)

print('Average precision-recall score: {0:0.2f}'.format(
      average_precision))
plt.figure()
plt.step(recall, precision, color='b', alpha=0.3,
         where='post')
plt.fill_between(recall, precision, step='post', alpha=0.3,
                 color='b')

plt.xlabel('Recall')
plt.ylabel('Precision')
plt.ylim([0.0, 1.2])
plt.xlim([0.0, 1.2])
plt.title('2-class Precision-Recall curve: AP={0:0.2f}'.format(
          average_precision))

print ("roc auc score: ", roc_auc_score(trueValues, predValues))
```

```
Average precision-recall score: 0.99
roc auc score:  0.990632853506
```



The classifiers have been assessed and the most accurate has been chosen and it's metrics detailed. The classifier can now be saved to disk. The saved classifier model will then be loaded and validated with our validation set from earlier.

In [691]:
```
# modify this variablefor a local directory
savedModel = "C:\\Users\\user\\Documents\\******\\Enron\\OurModel.sav"
```

In [692]:
```
pickle.dump(LinearSVCpipeline, open(savedModel, 'wb')) # save the model to disk
```

In [693]:
```
assignmentModel = pickle.load(open(savedModel, 'rb')) # reload the saved model
```

In [694]:
```
print ("loaded pickle: ", assignmentModel) # details of the model
```

```
loaded pickle:  Pipeline(steps=[('vectorizer', TfidfVectorizer(analyzer='word', binary=False, decode_error='strict',
        dtype=<class 'numpy.int64'>, encoding='utf-8', input='content',
        lowercase=True, max_df=0.69, max_features=None, min_df=1,
        ngram_range=(1, 1), norm='l2', preprocessor=None, smooth_idf=T... penalty='l2', random_state=None, tol=0.000
1,
     verbose=0),
           cv=3, method='sigmoid'))])
```

Using the validation dataframe from earlier, the saved model can then be used to predict on the unseen data.

In [696]:
```
validationResult = assignmentModel.predict(validationSet.Body.values)
```

In [697]:

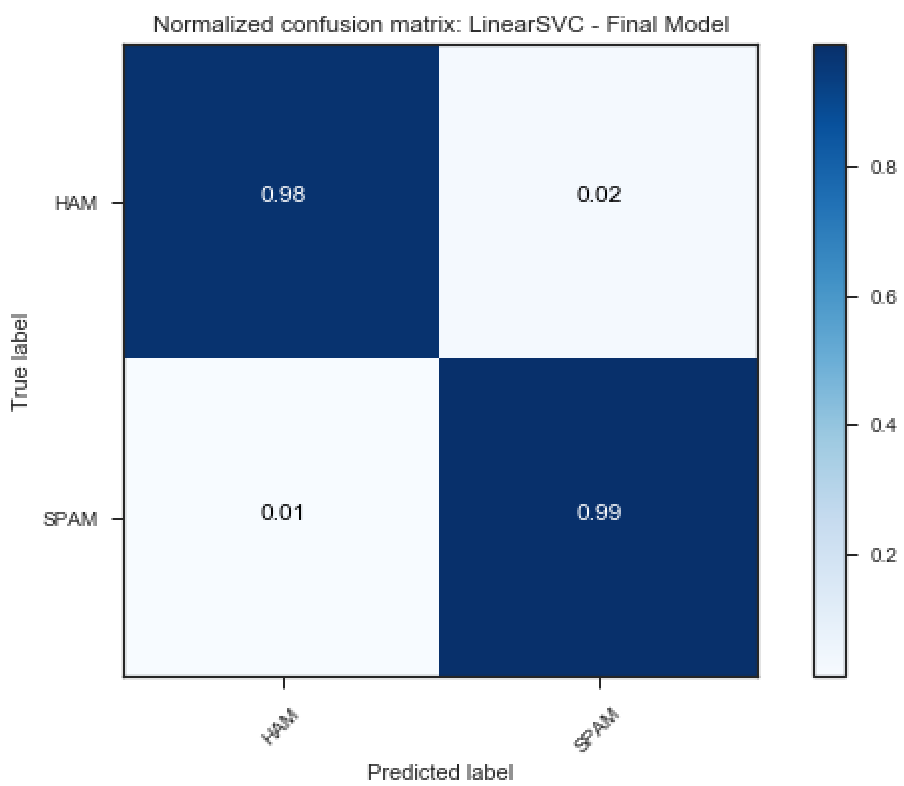
```
print ("--------------")
print ("Model score ", metrics.accuracy_score(validationSet.Classification.values, validationResult))
print ("--------------")
print (metrics.classification_report(validationSet.Classification.values, validationResult ))
print ("Mean classification accuracy")
print (np.mean(validationSet.Classification.values == validationResult))
```

```
--------------
Model score  0.985537190083
--------------
             precision    recall  f1-score   support

        HAM       0.99      0.98      0.99       243
       SPAM       0.98      0.99      0.99       241

avg / total       0.99      0.99      0.99       484

Mean classification accuracy
0.985537190083
```

In [698]:

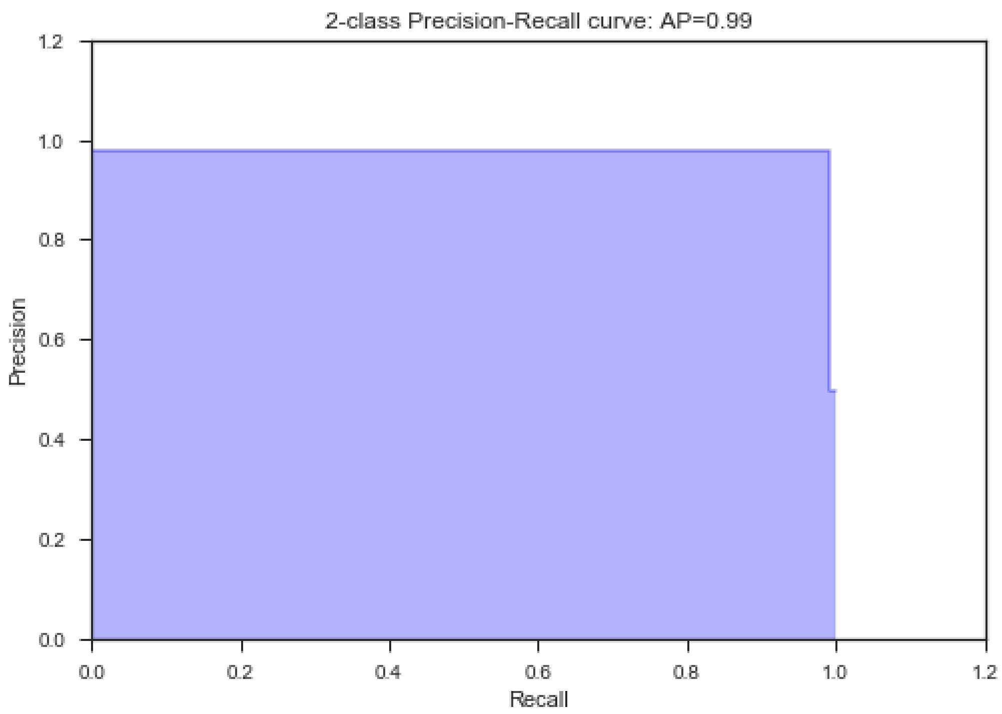
```
class_names = ["HAM", "SPAM"]
FinalModelMetrics = (metrics.confusion_matrix(validationSet.Classification.values, validationResult))
plt.figure()
plot_confusion_matrix(FinalModelMetrics, classes=class_names, normalize=True, title='Normalized confusion matrix: Line
arSVC - Final Model')
```

```
Normalized confusion matrix
[[ 0.97942387  0.02057613]
 [ 0.00829876  0.99170124]]
```



In [701]:
```
lb = LabelBinarizer()


trueValues = lb.fit_transform(validationSet.Classification.values)
LabelBinarizer(neg_label=0, pos_label=1, sparse_output=False)
predValues = lb.fit_transform(validationResult)
precision, recall, _ = metrics.precision_recall_curve(trueValues, predValues)
average_precision = average_precision_score(trueValues, predValues)

print('Average precision-recall score: {0:0.2f}'.format(
      average_precision))
plt.figure()
plt.step(recall, precision, color='b', alpha=0.3,
         where='post')
plt.fill_between(recall, precision, step='post', alpha=0.3,
                 color='b')

plt.xlabel('Recall')
plt.ylabel('Precision')
plt.ylim([0.0, 1.2])
plt.xlim([0.0, 1.2])
plt.title('2-class Precision-Recall curve: AP={0:0.2f}'.format(
          average_precision))

print ("ROC AUC score: ", roc_auc_score(trueValues, predValues))
```

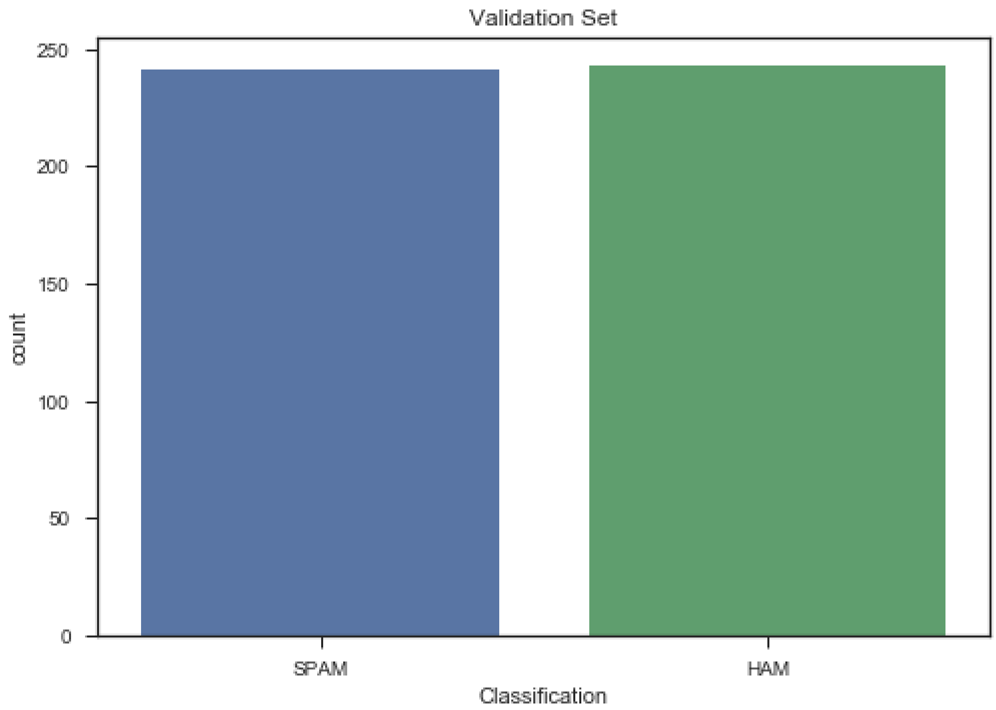
```
Average precision-recall score: 0.99
ROC AUC score:  0.985562556563
```



The classifier achieved significantly high results in comparison to some of the classifiers in our testing set. It could be said given a large enough dataset the results for the other classifiers could catch up. However, given the size of the validation set (below) and the test set evaluated earlier, it seems that the chosen classifier is still more accurate given its results thus far.

In [702]:
```
# Validation Set Stats
sns.countplot(validationSet.Classification).set_title("Validation Set")


validationBodyLength = []
for mails in (validationSet.Body.str.len()):
    validationBodyLength.append(mails)
validationMailSizes = pd.DataFrame({'Body':validationBodyLength})

print ("Mean validation mail body length: ", validationMailSizes.Body.mean())
print ("Total validation set size: ", validationMailSizes.Body.count())
```

```
Mean validation mail body length:  1451.1714876033059
Total validation set size:  484
```



Our final classifier obtained the following metrics:

- **Average precision-recall score:** 0.99
- **ROC AUC score:** 0.985562556563
- **Mean classification accuracy:** 0.985537190083

In [ ]:
```

```