Q3：SMS Spam Collection Dataset

In [1]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import nltk
import sklearn as sklearn
%matplotlib inline
import warnings
warnings.filterwarnings('ignore')
%config InlineBackend.figure_format = 'retina'
```

1. Import data

In [2]:
```
%cd /Users/Allen/Documents/ST4240/dataset
data = pd.read_csv("spam.csv",encoding='latin-1')
#Drop column and name change
data = data.drop(["Unnamed: 2", "Unnamed: 3", "Unnamed: 4"], axis=1)
data = data.rename(columns={"v1":"label", "v2":"text"})

from sklearn.preprocessing import LabelEncoder
le = sklearn.preprocessing.LabelEncoder()
le.fit(data["label"])
data["label"] = le.transform(data["label"])    #change the labels to 0 and 1
```

```
/Users/Allen/Documents/ST4240/dataset
```

2. Create training and testing set

In [3]:
```
from sklearn.model_selection import train_test_split
train,test = train_test_split(data, test_size = 0.3, random_state = 42,
                                                    shuffle=True, stratify= data["label"] )
#select balanced sample
X_train = train["text"]
X_test = test["text"]
y_train = train["label"]
y_test = test["label"]
#Separate the training set to "ham" and "spam"
train_ham = train.loc[train.label == 0]
train_spam = train.loc[train.label == 1]
```

3. Examine words in two categories

After converting all words to lower case, we use the RegexpTokenizer from nltk to tokenize words and remove all punctuations. After tokenising the words, we use WordNetLemmatizer as the stemming method to convert the words to their lemma and combine similar wordings. Lastly, we remove stopwords and natural numbers and create a list with all the remaining words.

In [4]:
```
from nltk.corpus import stopwords
from nltk.tokenize import RegexpTokenizer
tokenizer = RegexpTokenizer(r'\w+') #tokenize words while removing punctuations
from nltk.stem import WordNetLemmatizer
lemmatizer = WordNetLemmatizer() #to combine words of same lemma
```

In [5]:
```
hamword = []
for i in train_ham.text:
    words = i.lower()
    words = tokenizer.tokenize(words)
    for j in words:
        if j not in stopwords.words("english"):
            if not j.isdigit():
                j = lemmatizer.lemmatize(j)
                hamword.append(j)

spamword = []
for i in train_spam.text:
    words = i.lower()
    words = tokenizer.tokenize(words)
    for j in words:
        if j not in stopwords.words("english"):
            if not j.isdigit():
                j = lemmatizer.lemmatize(j)
                spamword.append(j)
```

Examine the top 10 words occurring in both "Spam" and "ham" messages in Xtrain

In [6]:
```
from collections import Counter
Counter(hamword).most_common(10)
```

Out[6]:
```
[('u', 747),
 ('gt', 233),
 ('lt', 230),
 ('get', 219),
 ('go', 198),
 ('ok', 189),
 ('call', 183),
 ('know', 179),
 ('ur', 170),
 ('got', 170)]
```

In [7]:
```
Counter(spamword).most_common(10)
```

Out[7]:
```
[('call', 250),
 ('å', 198),
 ('free', 166),
 ('u', 124),
 ('txt', 120),
 ('ur', 105),
 ('text', 104),
 ('stop', 89),
 ('mobile', 86),
 ('claim', 79)]
```

From the two tables we observe that there are some common terms like "u" and "call", but there are also words like "free", "text" and "stop" that tend to appear in spam messages.

4. Vectorizing Xtrain and Xtest

To prepare the training data for model building, we use vectorizer from sklearn.feature_extraction to convert Xtrain and Xtest to Compressed Sparse matrix. We use TfidfVectorizer with stopword, select"idf"=True to reduce the weights of frequently occurred words, so that they will have less impact in the model.

In [8]:
```
from sklearn.feature_extraction.text import TfidfVectorizer
vectorizer = TfidfVectorizer(stop_words='english',lowercase=True,use_idf=True)
```

In [9]:
```
Xtrain = vectorizer.fit_transform(X_train)
Xtest = vectorizer.transform(X_test) #use the fitted vectorizer to transform X_test
print(Xtrain.shape,Xtest.shape)
```

```
(3900, 6946) (1672, 6946)
```

5. Model Building

5.1 Benchmark model: Multinomial Naivebayes

For our benchmark model we build a Multinomail naivebayes model and examine the performance metrics.

Side note: From Sklearn documentation, we learnt that BernoulliNaiveBayes is also useful with short documents and binary features. By choosing ("binary" = True) and ("use_idf"=False) in TfidfVectorizer, we managed to obtain the transformed data as occurrence(0 or 1) instead of count. The prediction result is similar to MultinomialNB, and hence we stick to MultinomialNB to allow for easier comparisons among models.

In [10]:
```
from sklearn.metrics import accuracy_score,confusion_matrix,classification_report,auc
```

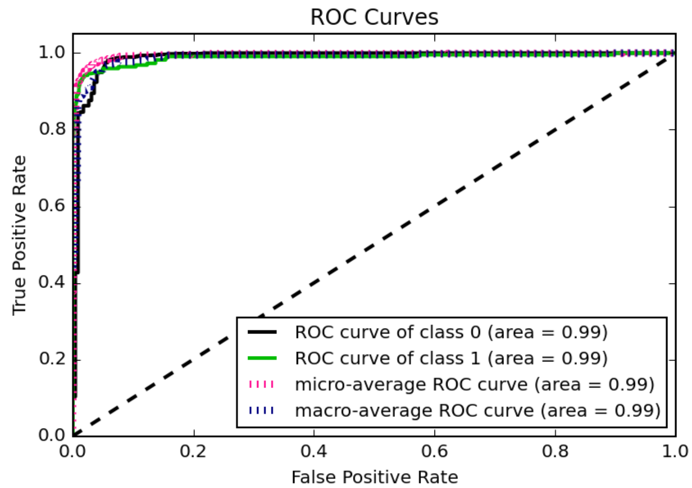
In [11]:
```
from sklearn.naive_bayes import MultinomialNB
NB = MultinomialNB()
NB.fit(Xtrain,y_train)
NB_pred = NB.predict(Xtest)
NB_pred_proba = NB.predict_proba(Xtest)
```

In [12]:
```
print ("prediciton Accuracy : %f" % accuracy_score(y_test, NB_pred))
```

```
prediciton Accuracy : 0.968301
```

In [13]:
```
#Plot the ROC curve to examine the performance of the model
import scikitplot as skplt
import matplotlib.pyplot as plt

skplt.metrics.plot_roc_curve(y_test, NB_pred_proba)
plt.show()
print ("AUC Score : %f" % sklearn.metrics.roc_auc_score(y_test, NB_pred_proba[:,1]))
```




```
AUC Score : 0.987751
```

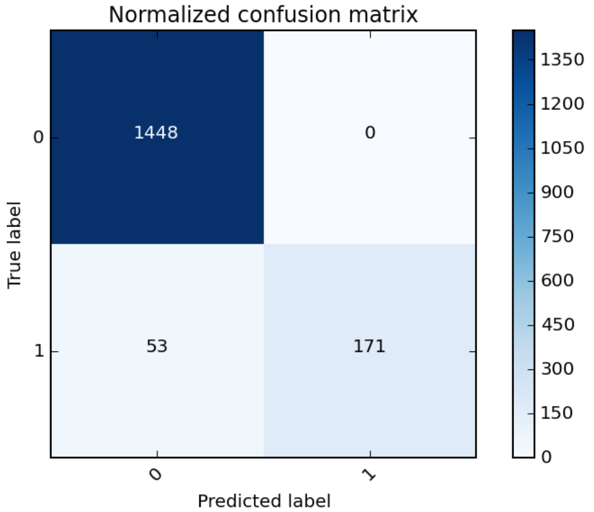
While both prediction accuracy(0.968301) and ROC curve(AUC=0.987751) suggest that the model performs very well in classifying the two messages, we continue to examine the confusion matrix and classification report to analyse the results closely. To visualize these two reports we use two function online. (see reference)

In [14]:

```
#Confusion Matrix
import itertools
def plot_confusion_matrix(cm, classes,
                          normalize=False,
                          title='Confusion matrix',
                          cmap=plt.cm.Blues):
    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 = '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')
plt.figure()
plot_confusion_matrix(confusion_matrix(y_test,NB_pred), classes=['0', '1'], normalize=Fal
se,
                      title='Normalized confusion matrix')
plt.show()
```

```
Confusion matrix, without normalization
[[1448    0]
 [  53  171]]
```



In [15]:
```
#Classification table
def plot_classification_report(cr, title='Classification report ', with_avg_total=False,
cmap=plt.cm.Blues):
    lines = cr.split('\n')
    classes = []
    plotMat = []
    for line in lines[2 : (len(lines) - 3)]:
        t = line.split()
        classes.append(t[0])
        v = [float(x) for x in t[1: len(t) - 1]]
        print(v)
        plotMat.append(v)
    if with_avg_total:
        aveTotal = lines[len(lines) - 1].split()
        classes.append('avg/total')
        vAveTotal = [float(x) for x in t[1:len(aveTotal) - 1]]
        plotMat.append(vAveTotal)
    plt.imshow(plotMat, interpolation='nearest', cmap=cmap)
    plt.title(title)
    plt.colorbar()
    x_tick_marks = np.arange(3)
    y_tick_marks = np.arange(len(classes))
    plt.xticks(x_tick_marks, ['precision', 'recall', 'f1-score'], rotation=45)
    plt.yticks(y_tick_marks, classes)
    plt.tight_layout()
    plt.ylabel('Classes')
    plt.xlabel('Measures')
print(classification_report(y_test, NB_pred, labels=['0', '1']))
```

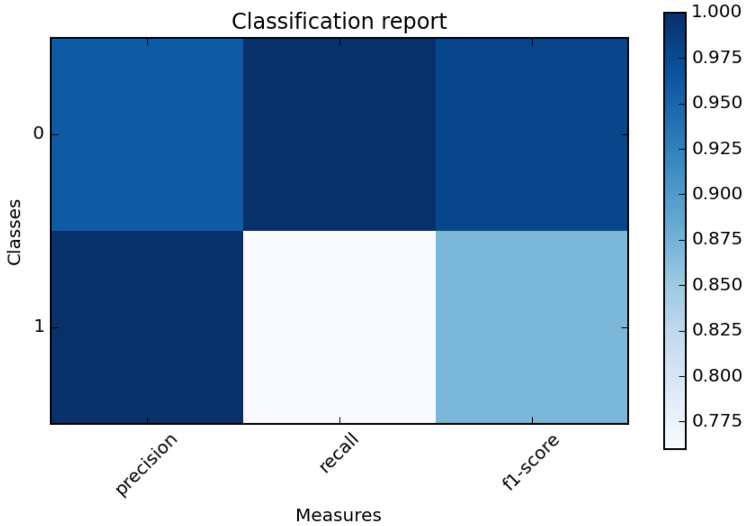
```
             precision    recall  f1-score   support

          0       0.96      1.00      0.98      1448
          1       1.00      0.76      0.87       224

avg / total       0.97      0.97      0.97      1672
```

In [16]:
```
plot_classification_report(classification_report(y_test, NB_pred, labels=['0', '1']))
```

```
[0.96, 1.0, 0.98]
[1.0, 0.76, 0.87]
```



In [17]:
```
#class ratio:
print("ham vs spam =",Counter(y_train)[0]/Counter(y_train)[1])
```

```
ham vs spam = 6.45697896749522
```

- From the confusion matrix, we see that all "ham" messages are correctly classfied while only 77% of the "spam" messages are correctly classified. The low accuracy of predicting "spam" is not reflected in the accuracy metric and ROC curve.
- From the classification report, we can tell that the precision of spam is perfect. However, we observe imbalance in the dataset as there are much more "ham" messages than "spam" messages. Therefore the low prediciton accuracy of "spam" messages is mitigated.
- Therefore, during model training, we should adjust the weights of the two categories. We now try to use different models to look for improvement of result.

In [31]:
```
#define a function for performance metrics
def model_eval(model,Xtrain,y_train, Xtest,y_test):
    pred = model.predict(Xtest)
    print ("Cross Validation score : ")
    print (cross_val_score(model, Xtrain, y_train, cv=5).mean())
    if not str(model)[:3] == "SGD":
        pred_proba = model.predict_proba(Xtest)
        pred_proba_c1 = pred_proba[:,1]
        print ("AUC Score : %f" % sklearn.metrics.roc_auc_score(y_test, pred_proba_c1))
    print ("prediciton Accuracy : %f" % accuracy_score(y_test, pred))
    print ("Confusion_matrix : ")
    print (confusion_matrix(y_test,pred))
    print ("classification report : ")
    print (classification_report(y_test, pred, labels=['0', '1']))
```

5.2 Logistic regression (with Cross Validation)

In [32]:
```
from sklearn.linear_model import LogisticRegressionCV
#"liblinear" is suitable for small dataset, use L1(Lasso) regularization to reduce dimens
ions, adjust weights
LRcv = LogisticRegressionCV(solver="liblinear",penalty = "l1",class_weight ="balanced")
LRcv.fit(Xtrain,y_train)
model_eval(LRcv,Xtrain, y_train, Xtest,y_test)
```

```
Cross Validation score :
0.979232057184
AUC Score : 0.981024
prediciton Accuracy : 0.979665
Confusion_matrix :
[[1434   14]
 [  20  204]]
classification report :
             precision    recall  f1-score   support

          0       0.99      0.99      0.99      1448
          1       0.94      0.91      0.92       224

avg / total       0.98      0.98      0.98      1672
```

5.3 Random Forest Classifier

In [33]:
```
#random forest works well when number of features is huge.
from sklearn.ensemble import RandomForestClassifier
from sklearn.cross_validation import cross_val_score
RF = RandomForestClassifier(n_estimators =100, max_features = "sqrt",bootstrap = True, oo
b_score=True,verbose=0,
                            class_weight = "balanced",random_state=42,max_depth = 40)
RF.fit(Xtrain,y_train)
print("RF.oob_score : %f" % RF.oob_score_)
model_eval(RF,Xtrain, y_train,Xtest,y_test)
```

```
RF.oob_score : 0.976154
Cross Validation score :
0.976671239408
AUC Score : 0.979422
prediciton Accuracy : 0.975478
Confusion_matrix :
[[1448    0]
 [  41  183]]
classification report :
             precision    recall  f1-score   support

          0       0.97      1.00      0.99      1448
          1       1.00      0.82      0.90       224

avg / total       0.98      0.98      0.97      1672
```

Random forest does not perform very well since this is a very sparse matrix, the effetiveness of separating nodes are not good enough since many words just occurred in few documents.

5.4 Gradient Boosting Classifier

In [34]:
```
from sklearn.ensemble import GradientBoostingClassifier
GBC = GradientBoostingClassifier(n_estimators=100, max_features = "sqrt", learning_rate=
0.25,
     max_depth=100, subsample= 0.8, random_state=42)

#create sample_weights array
sample_weights = [0.15 if x == 0 else 0.85 for x in y_train]
GBC.fit(Xtrain,y_train,sample_weight = sample_weights)
model_eval(GBC,Xtrain, y_train,Xtest,y_test)
```

```
Cross Validation score :
0.976927649664
AUC Score : 0.988724
prediciton Accuracy : 0.977871
Confusion_matrix :
[[1446    2]
 [  35  189]]
classification report :
             precision    recall  f1-score   support

          0       0.98      1.00      0.99      1448
          1       0.99      0.84      0.91       224

avg / total       0.98      0.98      0.98      1672
```

5.5 SGD Classifier

```
In [35]: from sklearn.linear_model import SGDClassifier
         #using SVM with loss="hinge" will automatically deal with the imbalance in the dataset
         SGD = SGDClassifier(loss="hinge", penalty="l2", alpha=0.0001,
                             l1_ratio=0.15, fit_intercept=True,
                             shuffle=True, learning_rate="optimal", n_iter= np.ceil(10**6 / Xtrain
         .shape[1]))
         SGD.fit(Xtrain,y_train)
         model_eval(SGD,Xtrain, y_train,Xtest,y_test)
```

```
Cross Validation score :
0.978460525715
prediciton Accuracy : 0.986244
Confusion_matrix :
[[1446    2]
 [  21  203]]
classification report :
             precision    recall  f1-score   support

          0       0.99      1.00      0.99      1448
          1       0.99      0.91      0.95       224

avg / total       0.99      0.99      0.99      1672
```

SVM works well even if dimension is greater than sample number. Results of logistic regression are not as good as svm under SGD, w/o LSA/L1/L2/elastic net regularisation.

6. SVD/ISA

In this session we try to reduce the number of dimensions since the number of features is huge.

```
In [28]: from sklearn.pipeline import make_pipeline
         from sklearn.decomposition import TruncatedSVD
         from sklearn.preprocessing import Normalizer
         svd = TruncatedSVD(n_components=100, random_state=42)   #dimension 100 as recommended by
          sklearn documentation
         lsa = make_pipeline(svd, Normalizer(copy=False))
         Xtrain_lsa = lsa.fit_transform(Xtrain)
         Xtest_lsa = lsa.transform(Xtest)
         print(svd.explained_variance_ratio_.sum())
```

```
0.252380864308
```

```
In [36]: #use Gradient Boosting on the transformed data
         GBC_lsa = GradientBoostingClassifier(n_estimators=100, max_features = "sqrt", learning_ra
         te=0.25,
              max_depth=20, subsample= 0.8, random_state=42)
         #create sample_weights array
         sample_weights = [0.15 if x == 0 else 0.85 for x in y_train]
         GBC_lsa.fit(Xtrain_lsa,y_train,sample_weight = sample_weights)
         model_eval(GBC_lsa,Xtrain_lsa, y_train,Xtest_lsa,y_test)
```

```
Cross Validation score :
0.974619954828
AUC Score : 0.979553
prediciton Accuracy : 0.976675
Confusion_matrix :
[[1434   14]
 [  25  199]]
classification report :
             precision    recall  f1-score   support

          0       0.98      0.99      0.99      1448
          1       0.93      0.89      0.91       224

avg / total       0.98      0.98      0.98      1672
```

The result shows that incorporating too much semantic information may not necessarily help with classification.

7. Model selection

- All models improved as compared to the benchmark model based on the performances metrics. Logistic regression with Lasso regularization achieves good result with simple method, however, the precision of "spam" is not as good as other models.
- Both random forest and gradient boosting give 100% precision rate for "spam", as well as high prediction accuracy. SVM using SGD on the other hand gives a higher prediction accuracy while incorrectly identify two "ham" as "spam".
- In our analysis, we should focus on precision of "spam" because we do not want to identity "ham" messages as "spam" messages in real life practice, while letting a small amount of "spam" escaping is acceptable. The cost of inaccurately identity a "ham" message as "spam" message should be higher than the other case.
- We choose GBC and perform a grid search to improve the prediction result.

In [25]:
```
from sklearn.model_selection import GridSearchCV
#for the first parameter, we try to look for the best n_estimators under learning_rate =
 0.1
param_test1 = {'n_estimators':range(50,151,10)}
gsearch1 = GridSearchCV(estimator = GradientBoostingClassifier(learning_rate=0.1,
                                   min_samples_leaf=10,max_depth=100,max_features='sqrt',

                                    subsample=0.8,random_state=42),
                       param_grid = param_test1, scoring='roc_auc',iid=False,cv=5)
gsearch1.fit(Xtrain,y_train)
gsearch1.grid_scores_, gsearch1.best_params_, gsearch1.best_score_
```

Out[25]:
```
([mean: 0.98560, std: 0.00644, params: {'n_estimators': 50},
  mean: 0.98604, std: 0.00659, params: {'n_estimators': 60},
  mean: 0.98639, std: 0.00559, params: {'n_estimators': 70},
  mean: 0.98707, std: 0.00507, params: {'n_estimators': 80},
  mean: 0.98766, std: 0.00461, params: {'n_estimators': 90},
  mean: 0.98733, std: 0.00472, params: {'n_estimators': 100},
  mean: 0.98752, std: 0.00387, params: {'n_estimators': 110},
  mean: 0.98757, std: 0.00389, params: {'n_estimators': 120},
  mean: 0.98774, std: 0.00385, params: {'n_estimators': 130},
  mean: 0.98757, std: 0.00376, params: {'n_estimators': 140},
  mean: 0.98742, std: 0.00384, params: {'n_estimators': 150}],
 {'n_estimators': 130},
 0.98774282374795208)
```

In [26]:
```
#We then use the best estimated n_estimators(130) and search for the best max_depth
param_test2 = {'max_depth':range(15,51,5)}
gsearch2 = GridSearchCV(estimator = GradientBoostingClassifier(learning_rate=0.1, n_estim
ators=130,
                                    min_samples_leaf=10, max_features='sqrt',
                                        subsample=0.8, random_state=42),
                   param_grid = param_test2, scoring='roc_auc',iid=False, cv=5)
gsearch2.fit(Xtrain,y_train)
gsearch2.grid_scores_, gsearch2.best_params_, gsearch2.best_score_
```

Out[26]:
```
([mean: 0.98646, std: 0.00468, params: {'max_depth': 15},
  mean: 0.98724, std: 0.00497, params: {'max_depth': 20},
  mean: 0.98763, std: 0.00455, params: {'max_depth': 25},
  mean: 0.98718, std: 0.00424, params: {'max_depth': 30},
  mean: 0.98819, std: 0.00323, params: {'max_depth': 35},
  mean: 0.98729, std: 0.00361, params: {'max_depth': 40},
  mean: 0.98767, std: 0.00387, params: {'max_depth': 45},
  mean: 0.98761, std: 0.00359, params: {'max_depth': 50}],
 {'max_depth': 35},
 0.98819316448034411)
```

In [27]:
```
#min_samples_split and min_samples_leaf since these two parameters are related
param_test3 = {'min_samples_split':range(100,301,50), 'min_samples_leaf':range(3,24,10)}
gsearch3 = GridSearchCV(estimator = GradientBoostingClassifier(learning_rate=0.1, n_estim
ators=130,
                                    max_depth=35,max_features='sqrt',
                                        subsample=0.8, random_state=42),
                       param_grid = param_test3, scoring='roc_auc',iid=False, cv=5)
gsearch3.fit(Xtrain,y_train)
gsearch3.grid_scores_, gsearch3.best_params_, gsearch3.best_score_
```

Out[27]:
```
([mean: 0.99043, std: 0.00242, params: {'min_samples_split': 100, 'min_samples_leaf':
3},
  mean: 0.99044, std: 0.00281, params: {'min_samples_split': 150, 'min_samples_leaf':
3},
  mean: 0.99044, std: 0.00281, params: {'min_samples_split': 200, 'min_samples_leaf':
3},
  mean: 0.99044, std: 0.00281, params: {'min_samples_split': 250, 'min_samples_leaf':
3},
  mean: 0.99044, std: 0.00281, params: {'min_samples_split': 300, 'min_samples_leaf':
3},
  mean: 0.98616, std: 0.00728, params: {'min_samples_split': 100, 'min_samples_leaf': 1
3},
  mean: 0.98616, std: 0.00728, params: {'min_samples_split': 150, 'min_samples_leaf': 1
3},
  mean: 0.98616, std: 0.00728, params: {'min_samples_split': 200, 'min_samples_leaf': 1
3},
  mean: 0.98616, std: 0.00728, params: {'min_samples_split': 250, 'min_samples_leaf': 1
3},
  mean: 0.98616, std: 0.00728, params: {'min_samples_split': 300, 'min_samples_leaf': 1
3},
  mean: 0.96934, std: 0.00844, params: {'min_samples_split': 100, 'min_samples_leaf': 2
3},
  mean: 0.96934, std: 0.00844, params: {'min_samples_split': 150, 'min_samples_leaf': 2
3},
  mean: 0.96934, std: 0.00844, params: {'min_samples_split': 200, 'min_samples_leaf': 2
3},
  mean: 0.96934, std: 0.00844, params: {'min_samples_split': 250, 'min_samples_leaf': 2
3},
  mean: 0.96934, std: 0.00844, params: {'min_samples_split': 300, 'min_samples_leaf': 2
3}],
 {'min_samples_leaf': 3, 'min_samples_split': 150},
 0.99043617398745598)
```

In [28]:
```
#max_features
param_test4 = {'max_features':range(40,131,10)}
gsearch4 = GridSearchCV(estimator = GradientBoostingClassifier(learning_rate=0.1, n_estim
ators=130,
                                    max_depth=35, min_samples_leaf =3, min_samples_split
=150,
                                            subsample=0.8, random_state=42),
                       param_grid = param_test4, scoring='roc_auc',iid=False, cv=5)
gsearch4.fit(Xtrain,y_train)
gsearch4.grid_scores_, gsearch4.best_params_, gsearch4.best_score_
```

Out[28]:
```
([mean: 0.99178, std: 0.00184, params: {'max_features': 40},
  mean: 0.99080, std: 0.00271, params: {'max_features': 50},
  mean: 0.99139, std: 0.00262, params: {'max_features': 60},
  mean: 0.99109, std: 0.00196, params: {'max_features': 70},
  mean: 0.99036, std: 0.00245, params: {'max_features': 80},
  mean: 0.98991, std: 0.00251, params: {'max_features': 90},
  mean: 0.99018, std: 0.00292, params: {'max_features': 100},
  mean: 0.98965, std: 0.00285, params: {'max_features': 110},
  mean: 0.98981, std: 0.00282, params: {'max_features': 120},
  mean: 0.99080, std: 0.00275, params: {'max_features': 130}],
 {'max_features': 40},
 0.99178147000198291)
```

In [29]:
```
#subsample
param_test5 = {'subsample':[0.6,0.7,0.75,0.8,0.85,0.9]}
gsearch5 = GridSearchCV(estimator = GradientBoostingClassifier(learning_rate=0.1, n_estim
ators=130,
                                    max_depth=35, min_samples_leaf =3, min_samples_split
=150,
                                                max_features=40, random_state=42),
                       param_grid = param_test5, scoring='roc_auc',iid=False, cv=5)
gsearch5.fit(Xtrain,y_train)
gsearch5.grid_scores_, gsearch5.best_params_, gsearch5.best_score_
```

Out[29]:
```
([mean: 0.99195, std: 0.00240, params: {'subsample': 0.6},
  mean: 0.99280, std: 0.00185, params: {'subsample': 0.7},
  mean: 0.99277, std: 0.00237, params: {'subsample': 0.75},
  mean: 0.99178, std: 0.00184, params: {'subsample': 0.8},
  mean: 0.99180, std: 0.00250, params: {'subsample': 0.85},
  mean: 0.99236, std: 0.00273, params: {'subsample': 0.9}],
 {'subsample': 0.7},
 0.99279687966354635)
```

8. Final model

Now we use all the parameter estimated in the model. Reduce "learning_rate" by half and double "n_estimators".

In [37]:
```
GBC2 = GradientBoostingClassifier(learning_rate=0.05, n_estimators=260,max_depth=35, min_
samples_leaf =3,
               min_samples_split =150, max_features=40, subsample=0.7, random_state=42)
sample_weights = [0.15 if x == 0 else 0.85 for x in y_train]
GBC2.fit(Xtrain,y_train,sample_weight = sample_weights)
model_eval(GBC2,Xtrain, y_train,Xtest,y_test)
```

```
Cross Validation score :
0.981027590657
AUC Score : 0.984558
prediciton Accuracy : 0.988636
Confusion_matrix :
[[1448    0]
 [  19  205]]
classification report :
             precision    recall  f1-score   support

          0       0.99      1.00      0.99      1448
          1       1.00      0.92      0.96       224

avg / total       0.99      0.99      0.99      1672
```

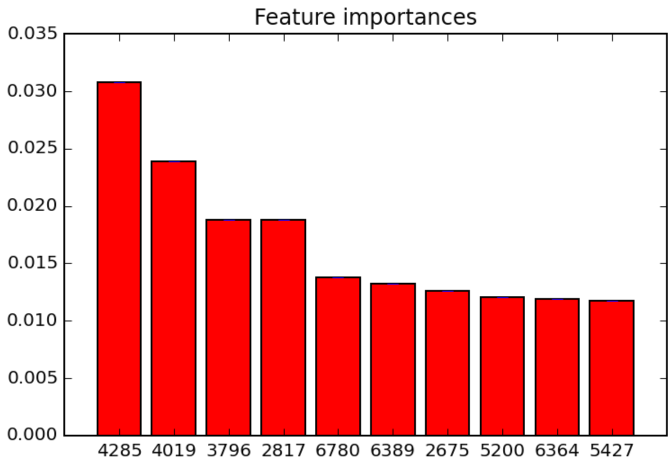
Both AUC score and prediciton Accuracy increase from original model

plot top10 feature importance

In [31]:
```
importances = GBC2.feature_importances_
std = np.std([GBC2.feature_importances_ for tree in GBC2.estimators_],axis=0)
```

In [32]:
```
#top 10 indices:
indices = np.argsort(importances)[::-1][0:10]
feature_names = vectorizer.get_feature_names()
print ("top10words : ")
for i in range(10):
    print (indices[i],feature_names[indices[i]])
plt.figure()
plt.title("Feature importances")
plt.bar(range(10), importances[indices],
       color="r", yerr=std[indices], align="center")
plt.xticks(range(10), indices)
plt.xlim([-1, 10])
plt.show()
```

```
top10words :
4285 new
4019 message
3796 lost
2817 girls
6780 won
6389 uk
2675 free
5200 ringtone
6364 txt
5427 sexy
```



Further Improvements

- Study the numbers in the features, compare the results after removing the numbers.
- Study the wrongly classfied messages for further insights.

Reference

- confusion matrix http://scikit-learn.org/stable/auto_examples/model_selection/plot_confusion_matrix.html#sphx-glr-auto-examples-model-selection-plot-confusion-matrix-py (http://scikit-learn.org/stable/auto_examples/model_selection/plot_confusion_matrix.html#sphx-glr-auto-examples-model-selection-plot-confusion-matrix-py)
- classification table https://stackoverflow.com/questions/28200786/how-to-plot-scikit-learn-classification-report?noredirect=1&lq=1 (https://stackoverflow.com/questions/28200786/how-to-plot-scikit-learn-classification-report?noredirect=1&lq=1)
- plot feature importance http://scikit-learn.org/stable/auto_examples/ensemble/plot_forest_importances.html (http://scikit-learn.org/stable/auto_examples/ensemble/plot_forest_importances.html)