In [31]:
```
# Initialize Otter
import otter
grader = otter.Notebook("projB2.ipynb")
```

Project B2: Spam Detection

In [32]:
```
# Run this cell to suppress all FutureWarnings
import warnings
warnings.filterwarnings("ignore", category=FutureWarning)
```

In [33]:
```
import numpy as np
import pandas as pd

import matplotlib.pyplot as plt
%matplotlib inline

import seaborn as sns
sns.set(style = "whitegrid",
        color_codes = True,
        font_scale = 1.5)
```

Setup and Recap

Here we will provide a summary of Project B1 to remind you of how we cleaned the data, explored it, and implemented methods that are going to be useful for building your own model.

Loading and Cleaning Data

Remember that in email classification, our goal is to classify emails as spam or not spam (referred to as "ham") using features generated from the text in the email.

The dataset consists of email messages and their labels (0 for ham, 1 for spam). Your labeled training dataset contains 8,348 labeled examples, and the unlabeled test set contains 1,000 unlabeled examples.

Run the following cell to load in the data into a `DataFrame`.

The `train` DataFrame contains labeled data that you will use to train your model. It contains four columns:

1. `id` : An identifier for the training example.
2. `subject` : The subject of the email.
3. `email` : The text of the email.
4. `spam` : 1 if the email is spam, 0 if the email is ham (not spam).

The `test` DataFrame contains 1,000 unlabeled emails. You will predict labels for these emails and submit your predictions to the autograder for evaluation.

In [34]:
```
import zipfile
with zipfile.ZipFile('spam_ham_data.zip') as item:
    item.extractall()
```

In [35]:
```
original_training_data = pd.read_csv('train.csv')
test = pd.read_csv('test.csv')

# Convert the emails to lower case as a first step to processing the text
original_training_data['email'] = original_training_data['email'].str.lower(
test['email'] = test['email'].str.lower()

original_training_data.head()
```

Out[35]:

	id	subject	email	spam
0	0	Subject: A&L Daily to be auctioned in bankrupt...	url: http://boingboing.net/#85534171\n date: n...	0
1	1	Subject: Wired: "Stronger ties between ISPs an...	url: http://scriptingnews.userland.com/backiss...	0
2	2	Subject: It's just too small ...	<html>\n <head>\n </head>\n <body>\n <font siz...	1
3	3	Subject: liberal defnitions\n	depends on how much over spending vs. how much...	0
4	4	Subject: RE: [ILUG] Newbie seeks advice - Suse...	hehe sorry but if you hit caps lock twice the ...	0

Feel free to explore the dataset above along with any specific spam and ham emails that interest you. Keep in mind that our data may contain missing values, which are handled in the following cell.

In [36]:
```
# Fill any missing or NAN values
print('Before imputation:')
print(original_training_data.isnull().sum())
original_training_data = original_training_data.fillna('')
print('------------')
print('After imputation:')
print(original_training_data.isnull().sum())
```

```
Before imputation:
id         0
subject    6
email      0
spam       0
dtype: int64
------------
After imputation:
id         0
subject    0
email      0
spam       0
dtype: int64
```

Training/Validation Split

Recall that the training data we downloaded is all the data we have available for both training models and **validating** the models that we train. We therefore split the training data into separate training and validation datsets. You

In [37]:
```
# This creates a 90/10 train-validation split on our labeled data.
from sklearn.model_selection import train_test_split
train, val = train_test_split(original_training_data, test_size = 0.1, rando

# We must do this in order to preserve the ordering of emails to labels for
train = train.reset_index(drop = True)
```

Feature Engineering

In order to train a logistic regression model, we need a numeric feature matrix $\mathbb{X}$ and a vector of corresponding binary labels $\mathbb{Y}$.

In [38]:
```
def words_in_texts(words, texts):
    '''
    Args:
        words (list): words to find
        texts (Series): strings to search in

    Returns:
        A 2D NumPy array of 0s and 1s with shape (n, p) where n is the
        number of texts and p is the number of words.
    '''
    import numpy as np
    indicator_array = 1 * np.array([texts.str.contains(word) for word in wor
    return indicator_array
```

Run the following cell to see how the function works on some dummy text.

In [39]:
```
words_in_texts(['hello', 'bye', 'world'], pd.Series(['hello', 'hello worldhe
```

```
Out[39]: array([[1, 0, 0],
               [1, 0, 1]])
```

EDA and Basic Classification

In Project B1, we proceeded to visualize the frequency of different words for both spam and ham emails, and used `words_in_texts(words, train['email'])` to directly to train a classifier. We also provided a simple set of 5 words that might be useful as features to distinguish spam/ham emails.

We then built a model using the using the `LogisticRegression` classifier from `sklearn`.

Run the following cell to see the performance of a simple model using these words and the `train` dataframe.

```
In [40]: some_words = ['drug', 'bank', 'prescription', 'memo', 'private']

         X_train = words_in_texts(some_words, train['email'])
         Y_train = np.array(train['spam'])

         X_train[:5], Y_train[:5]
```

```
Out[40]: (array([[0, 0, 0, 0, 0],
                 [0, 0, 0, 0, 0],
                 [0, 0, 0, 0, 0],
                 [0, 0, 0, 0, 0],
                 [0, 0, 0, 1, 0]]),
          array([0, 0, 0, 0, 0]))
```

```
In [41]: from sklearn.linear_model import LogisticRegression

         model = LogisticRegression(solver = 'lbfgs')
         model.fit(X_train, Y_train)

         training_accuracy = model.score(X_train, Y_train)
         print("Training Accuracy: ", training_accuracy)
```

```
Training Accuracy:  0.7576201251164648
```

Evaluating Classifiers

In our models, we are evaluating accuracy on the training set, which may provide a misleading accuracy measure. In Project B1, we calculated various metrics to lead us to consider more ways of evaluating a classifier, in addition to overall accuracy. Below is a reference to those concepts.

Presumably, our classifier will be used for **filtering**, i.e. preventing messages labeled `spam` from reaching someone's inbox. There are two kinds of errors we can make:

- **False Positive (FP)**: A ham email gets flagged as spam and filtered out of the inbox.
- **False Negative (FN)**: A spam email gets mislabeled as ham and ends up in the inbox.

To be clear, we label spam emails as 1 and ham emails as 0. These definitions depend both on the true labels and the predicted labels. False positives and false negatives may be of differing importance, leading us to consider more ways of evaluating a classifier, in addition to overall accuracy:

Precision: Measures the proportion $\frac{\text{TP}}{\text{TP} + \text{FP}}$ of emails flagged as spam that are actually spam.

Recall: Measures the proportion $\frac{\text{TP}}{\text{TP} + \text{FN}}$ of spam emails that were correctly flagged as spam.

False positive rate: Measures the proportion $\frac{\text{FP}}{\text{FP} + \text{TN}}$ of ham emails that were incorrectly flagged as spam.

The below graphic (modified slightly from Wikipedia) may help you understand precision and recall visually:

Note that a True Positive (TP) is a spam email that is classified as spam, and a True Negative (TN) is a ham email that is classified as ham.

Moving Forward - Building Your Own Model

With this in mind, it is now your task to make the spam filter more accurate. In order to get full credit on the accuracy part of this assignment, you must get at least **85%** accuracy on both the train and test set (see Q4 for the partial credit breakdown). To see your accuracy on the test set, you will use your classifier to predict every email in the `test` DataFrame and upload your predictions to Gradescope.

Gradescope limits you to three submissions per day. You will be able to see your accuracy on the entire test set when submitting to Gradescope.

Here are some ideas for improving your model:

1. Finding better features based on the email text. Some example features are:
   A. Number of characters in the subject / body
   B. Number of words in the subject / body

    C. Use of punctuation (e.g., how many '!'s were there?)
    D. Number / percentage of capital letters
    E. Whether the email is a reply to an earlier email or a forwarded email
2. Finding better (and/or more) words to use as features. Which words are the best at distinguishing emails? This requires digging into the email text itself.
3. Better data processing. For example, many emails contain HTML as well as text. You can consider extracting out the text from the HTML to help you find better words. Or, you can match HTML tags themselves, or even some combination of the two.
4. Model selection. You can adjust parameters of your model (e.g. the penalty type, the regularization parameter, or any arguments in `LogisticRegression`) to achieve higher accuracy. Recall that you should use cross-validation to do feature and model selection properly! Otherwise, you will likely overfit to your training data.
    A. We have imported `GridSearchCV` for you. You may use sklearn's `GridSearchCV` (documentation) class to perform cross-validation. You do not need to code your own CV from scratch, though you are welcome to do so.

Here's an example of how to use GridSearchCV. Suppose we wanted to experiment with 3 different solvers (numerical methods for optimizing the mode) models for a LogisticRegression Model `lr_model`.

1. We could define a dictionary specifying the hyperparameters and the specific values we want to try out like so: `parameters = {'solver':[{‘lbfgs’, ‘liblinear’, ‘newton-cg’, 'saga']}`.
2. Running `grid = GridSearchCV(estimator=lr_model, param_grid=parameters)` would give us a model for each combination of hyperparameters we are testing - in this case, just 4 models.
3. We fit each model to some training data `X_train` and `Y_train` like so, `grid_result = grid.fit(X_train, Y_train)`
4. Indexing into `grid_result.cv_results_` with a particular metric (in this case, `mean_test_score`), we get an array with the scores corresponding to each of the models. `grid_result.cv_results_['mean_test_score']`.

Feel free to experiment with other hyperparameters and metrics as well, the documentation is your friend!

You may use whatever method you prefer in order to create features, but **you are not allowed to import any external feature extraction libraries**. In addition, **you are only allowed to train logistic regression models**. No decision trees, random forests, k-nearest-neighbors, neural nets, etc.

We have not provided any code to do this, so feel free to create as many cells as you need in order to tackle this task. However, answering questions 1, 2, and 3 should help guide you.

Note: You may want to use your **validation data** to evaluate your model and get a better sense of how it will perform on the test set. However, that you may overfit to your validation set if you try to optimize your validation accuracy too much. Alternatively, you can perform cross-validation on the entire training set.

In [42]:
```
from collections import Counter
import re
```

In [43]:
```
train['email_len'] = train['email'].str.len()
train['num_cap_letter'] = train['email'].apply(lambda sf: sum(1 for x in sf
train['num_special_char'] = train['email'].apply(lambda sf: sum(1 for c in s
train['reply'] = train['subject'].apply(lambda sf: 1 if 'RE' in sf else 0)
train['forward'] = train['subject'].apply(lambda sf: 1 if 'FW' in sf else 0)


spam_emails = train[train['spam'] == 1]['email']
ham_emails = train[train['spam'] == 0]['email']

# Initialize counters
common_spam = Counter()
common_ham = Counter()

# Update counters for each email
for email in spam_emails:
    common_spam.update(re.findall(r'\b\w+\b', email))

for email in ham_emails:
    common_ham.update(re.findall(r'\b\w+\b', email))

# Get most common words
most_common_spam_words = [word for word, _ in common_spam.most_common(50)]
most_common_ham_words = [word for word, _ in common_ham.most_common(10)]

spam_words_not_in_ham = [word for word in most_common_spam_words if word not

#adding these spam words not in ham, to the features we used in B1

final_words = ['drug', 'bank', 'prescription', 'memo', 'private'] + list(spa

word_features = words_in_texts(final_words, train['email'])
```

In [44]:
```
X_train = np.concatenate((train[['email_len', 'num_cap_letter', 'num_special
y_train = train['spam'].values

m1 = LogisticRegression(solver = "lbfgs", max_iter = 1000)

m1.fit(X_train, y_train)
```

Out[44]:
```
▾ LogisticRegression
LogisticRegression(max_iter=1000)
```

In [45]:
```
training_accuracy = m1.score(X_train, y_train)
```

In [46]:
```
training_accuracy
```

Out[46]:
```
0.8676959936110742
```

In [47]:
```
#THIS IS FOR THE TEST
# You may find it helpful to look through the rest of the questions first!


#defining new features

test['email_len'] = test['email'].str.len()
test['num_cap_letter'] = test['email'].apply(lambda sf: sum(1 for x in sf if
test['num_special_char'] = test['email'].apply(lambda sf: sum(1 for c in sf
test['subject'] = test['subject'].fillna('')
test['reply'] = test['subject'].apply(lambda sf: 1 if 'RE' in sf else 0)
test['forward'] = test['subject'].apply(lambda sf: 1 if 'FW' in sf else 0)


# Initialize counters
common_spam = Counter()
common_ham = Counter()

# Update counters for each email
for email in spam_emails:
    common_spam.update(re.findall(r'\b\w+\b', email))

for email in ham_emails:
    common_ham.update(re.findall(r'\b\w+\b', email))

# Get most common words
most_common_spam_words = [word for word, _ in common_spam.most_common(50)]
most_common_ham_words = [word for word, _ in common_ham.most_common(10)]

spam_words_not_in_ham = [word for word in most_common_spam_words if word not

#adding these spam words not in ham, to the features we used in B1

final_words = ['drug', 'bank', 'prescription', 'memo', 'private'] + list(spa

word_features_test = words_in_texts(final_words, test['email'])
```

In [48]:
```
X_test = np.concatenate((test[['email_len', 'num_cap_letter', 'num_special_c
```

Question 1

In this following cell, describe the process of improving your model. You should use at least 2-3 sentences each to address the follow questions:

1. How did you find better features for your model?
2. What did you try that worked or didn't work?
3. What was surprising in your search for good features?

I found better features by experimenting, and utilizing the hints by looking at features like email length, the number of capital letters, special characters, and whether the email was a reply or forward. Additionally, by adjusting my method of identifying common words in the spam emails, I was able to reduce memory consumption and complete the task without killing the kernel. I initially looped through each email and checked the frequency of that word, and it was resource intensive and consistently killing my kernel. By adjusting my approach to using a counter, I was able to efficiently parse the dataset and pull out the 30 most used spam words in this case.

Exploratory Data Analysis

In the cell below, show a visualization that you used to select features for your model.

Include:

1. A plot showing something meaningful about the data that helped you during feature selection, model selection, or both.
2. Two or three sentences describing what you plotted and its implications with respect to your features.

Feel free to create as many plots as you want in your process of feature selection, but select only one for the response cell below.

You should not just produce an identical visualization to Question 3 in Project B1. For this section, we'd like you to go beyond the analysis you performed in Project B1. Choose some plot other than the 1-dimensional distribution of some quantity for spam and ham emails. In particular, do not produce a bar plot of proportions like you created in Question 3 of Project B1. Any other plot is acceptable, **as long as it comes with thoughtful commentary.** Here are some ideas:

1. Consider the correlation between multiple features (look up correlation plots and `sns.heatmap`).
2. Try to show redundancy in a group of features (e.g. `body` and `html` might co-occur relatively frequently, or you might be able to design a feature that captures all

html tags and compare it to these).
3. Visualize which words have high or low values for some useful statistic.
4. Visually depict whether spam emails tend to be wordier (in some sense) than ham emails.

Question 2a

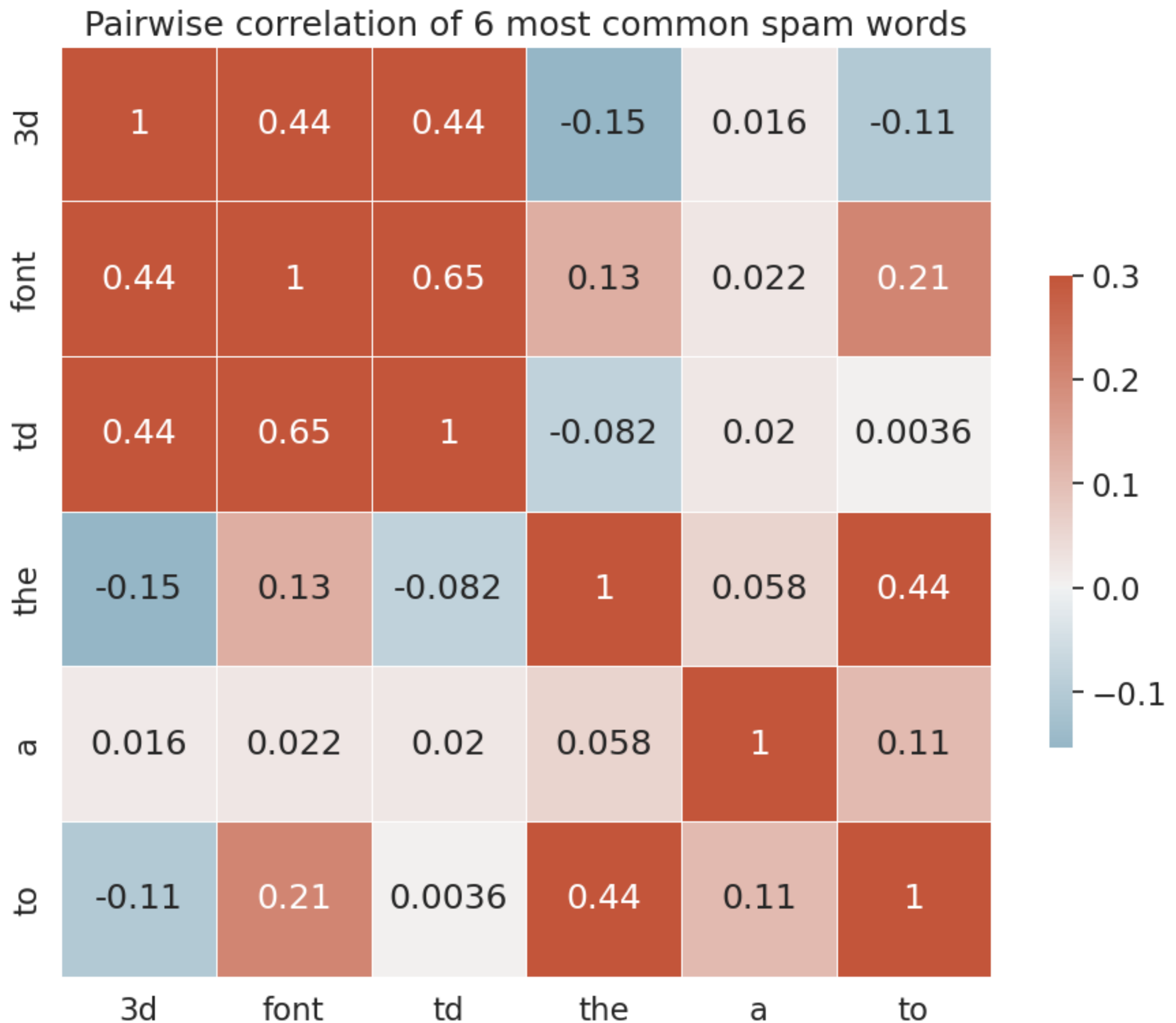
Generate your visualization in the cell below.

In [49]:
```
most_common_spam_words
```

Out[49]:
```
['3d',
 'font',
 'td',
 'the',
 'a',
 'to',
 'br',
 'size',
 'b',
 'and',
 'you',
 'tr',
 'of',
 'width',
 'face',
 'p',
 'color',
 'http',
 'nbsp',
 'com',
 'for',
 'your',
 'align',
 'in',
 '0',
 '1',
 '2',
 '20',
 'i',
 'this',
 'arial',
 'is',
 'center',
 'height',
 'www',
 'table',
 'border',
 'href',
 'that',
 'it',
 'or',
 'with',
 'be',
 'on',
 'verdana',
 'html',
 'helvetica',
 'are',
 'name',
 'style']
```

In [50]:
```
first_6_spam_words = most_common_spam_words[:6]

word_counts = pd.DataFrame(words_in_texts(first_6_spam_words, spam_emails),

corr = word_counts.corr()
```

```python
f, ax = plt.subplots(figsize=(11, 9))

cmap = sns.diverging_palette(230, 20, as_cmap=True)

sns.heatmap(corr, cmap=cmap, vmax=.3, center=0,
            square=True, linewidths=.5, cbar_kws={"shrink": .5}, annot=True)

plt.title('Pairwise correlation of 6 most common spam words')
plt.show()
```




```
In [ ]:
```

Question 2b

Write your commentary in the cell below.

The relationship among some of the most common spam words showcases that many of these words have a likelihood of appearing together in the same model, which boads well for our training accuracy as we want to corectly identify words commonly found in spam as our features.

Question 3: ROC Curve

In most cases we won't be able to get 0 false positives and 0 false negatives, so we have to compromise. For example, in the case of cancer screenings, false negatives are comparatively worse than false positives — a false negative means that a patient might not discover that they have cancer until it's too late, whereas a patient can just receive another screening for a false positive.

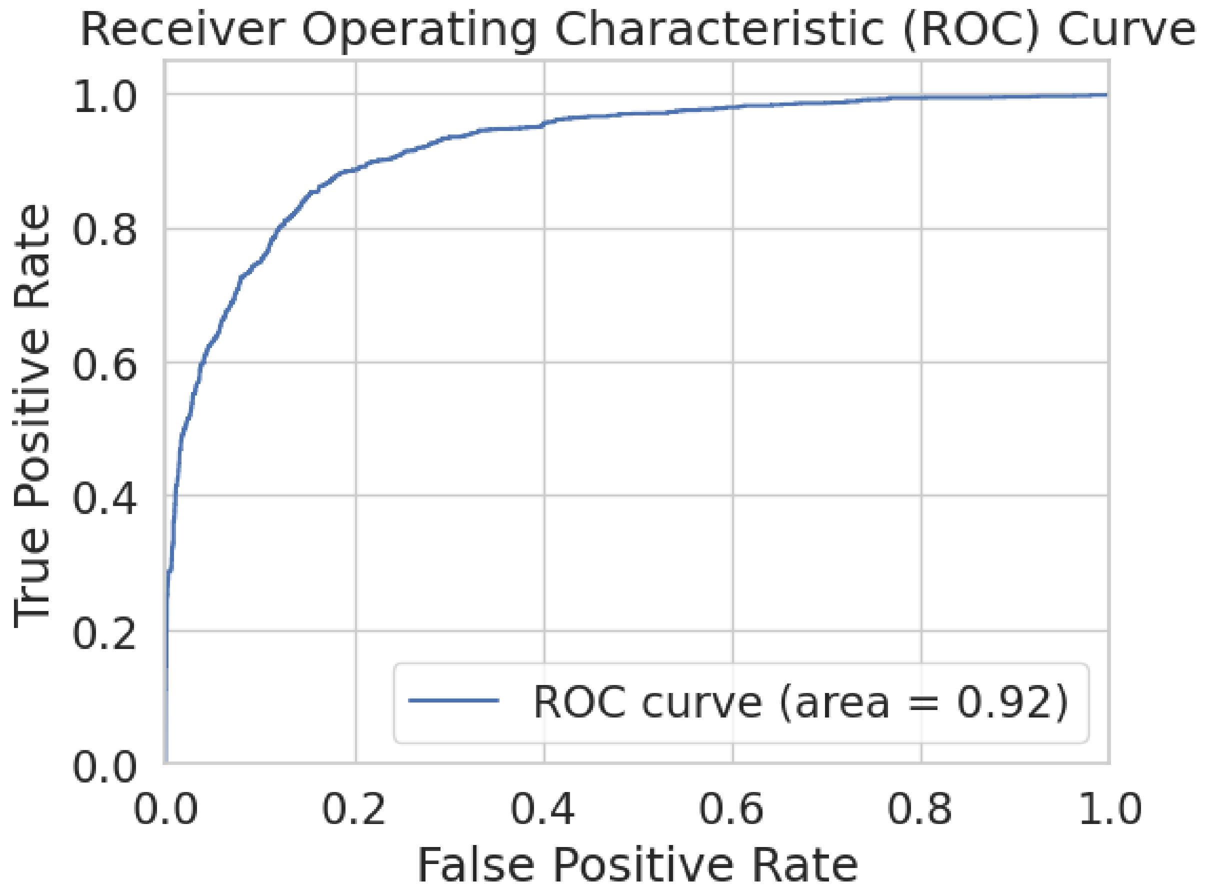
Recall that logistic regression calculates the probability that an example belongs to a certain class. Then, to classify an example we say that an email is spam if our classifier gives it ≥ 0.5 probability of being spam. However, *we can adjust that cutoff threshold*: we can say that an email is spam only if our classifier gives it ≥ 0.7 probability of being spam, for example. This is how we can trade off false positives and false negatives.

The Receiver Operating Characteristic (ROC) curve shows this trade off for each possible cutoff probability. In the cell below, plot a ROC curve for your final classifier (the one you use to make predictions for Gradescope) on the training data. Refer to Lecture 19 to see how to plot an ROC curve.

Hint: You'll want to use the `.predict_proba` method for your classifier instead of `.predict` to get probabilities instead of binary predictions.

In [51]:
```
from sklearn.metrics import roc_curve, auc

predicted_probabilities = m1.predict_proba(X_train)[:, 1]

fpr, tpr, thresholds = roc_curve(y_train, predicted_probabilities)

plt.figure()
plt.plot(fpr, tpr, label='ROC curve (area = %0.2f)' % auc(fpr, tpr))

plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic (ROC) Curve')
```

```
plt.legend(loc="lower right")
plt.show()
```



Question 4

Grading Scheme

Your grade for Question 4 will be based on your model's accuracy when making predictions on the training set, as well as your model's accuracy when making predictions on the test set. The tables below provide scoring guidelines. If your accuracy lies in a particular range, you will receive the number of points associated with that range.

Important: while your training accuracy can be checked at any time in this notebook, your test accuracy can only be checked by submitting your model's predictions to Gradescope. You may only submit to Gradescope 3 times a day. Plan ahead to make sure you have enough time to finetune your model! The thresholds are as follows:

Points	5	3	1.5	0
Training Accuracy	Above 85%	[80, 85)	[70, 80)	Below 70%

Points	10	6	3	0
Testing Accuracy	Above 85%	[80, 85)	[70, 80)	Below 70%

Question 4a: Train Predictions

Assign your predictions for the class of each datapoint in the training set `train` to the variable `train_predictions`.

In [52]:
```
train_predictions = m1.predict(X_train)
```

In [53]:
```
grader.check("q4a")
```

Out[53]:

q4a passed! 💯

Question 4b: Test Predictions

The following code will write your predictions on the test dataset to a CSV file. **You will need to submit this file to the "Project B2 Test Predictions" assignment on Gradescope to get credit for this question.**

Assign your predictions for the class of each datapoint in the test set `test` to a 1-dimensional array called `test_predictions`. **Please make sure you've saved your predictions to `test_predictions` as this is how part of your score for this question will be determined.**

Remember that if you've performed transformations or featurization on the training data, you must also perform the same transformations on the test data in order to make predictions. For example, if you've created features for the words "drug" and "money" on the training data, you must also extract the same features in order to use scikit-learn's `.predict(...)` method.

You may submit up to 3 times a day. If you have submitted 3 times on a day, you will need to wait until the next day for more submissions.

The provided tests check that your predictions are in the correct format, but you must additionally submit to Gradescope to evaluate your classifier accuracy.

In [54]:
```
test_predictions = m1.predict(X_test)
```

In [55]:
```
grader.check("q4b")
```

Out[55]:

q4b passed! 💯

The following cell generates a CSV file with your predictions. **You must submit this CSV file to the "Project B2 Test Predictions" assignment on Gradescope to get credit for this question.** There are a maximum of 3 attempts per day of submitting to this assignment, so please use them wisely!

In [56]:
```
from datetime import datetime
from IPython.display import display, HTML

# Assuming that your predictions on the test set are stored in a 1-dimension
# test_predictions. Feel free to modify this cell as long you create a CSV i

# Construct and save the submission:
submission_df = pd.DataFrame({
    "Id": test['id'],
    "Class": test_predictions,
}, columns=['Id', 'Class'])
timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
filename = "submission_{}.csv".format(timestamp)
submission_df.to_csv(filename, index=False)

print('Created a CSV file: {}.'.format("submission_{}.csv".format(timestamp)
display(HTML("Download your test prediction <a href='" + filename + "' downl
print('You may now upload this CSV file to Gradescope for scoring.')#
```

```
Created a CSV file: submission_20230908_090711.csv.
```
Download your test prediction here.
```
You may now upload this CSV file to Gradescope for scoring.
```

Congratulations! You have finished Project B2!

Below, you will see two cells. Running the first cell will automatically generate a PDF of all questions that need to be manually graded, and running the second cell will automatically generate a zip with your autograded answers. **You are responsible for both the coding portion (the zip from Project B2) and the written portion (the PDF**

with from Project B2) to their respective Gradescope portals, and checking that they are the most recent copy or the copy you wish to submit (including plots and all written answers). The coding proportion should be submitted to Project B2 Coding as a single zip file, and the written portion should be submitted to Project B2 Written as a single pdf file. When submitting the written portion, please ensure you select pages appropriately. In addition, you must submit your test prediction in Q4b to **Project B2 Test Set Predictions** for the corresponding points.

If there are issues with automatically generating the PDF in the first cell, you can try downloading the notebook as a PDF by clicking on `File -> Save and Export Notebook As... -> PDF`. If that doesn't work either, you can manually take screenshots of your answers to the manually graded questions and submit those. Either way, **you are responsible for ensuring your submission follows our requirements, we will NOT be granting regrade requests for submissions that don't follow instructions.**

In [57]:
```
from otter.export import export_notebook
from os import path
from IPython.display import display, HTML
export_notebook("projB2.ipynb", filtering=True, pagebreaks=True)
if path.exists('projB2.pdf'):
    display(HTML("Download your PDF <a href='projB2.pdf' download>here</a>."
else:
    print("\n Pdf generation fails, please try the other methods described a
```

Download your PDF here.

Submission

Make sure you have run all cells in your notebook in order before running the cell below, so that all images/graphs appear in the output. The cell below will generate a zip file for you to submit. **Please save before exporting!**

In [ ]:
```
# Save your notebook first, then run this cell to export your submission.
grader.export(run_tests=True)
```