proj2b

March 8, 2023

[1]:
```
# Initialize Otter
import otter
grader = otter.Notebook("proj2b.ipynb")
```

1 Project 2B: Spam/Ham Classification - Build Your Own Model

1.1 Feature Engineering, Classification, Cross Validation

1.2 Due Date: Monday, August 8th, 11:59 PM PDT

Collaboration Policy

Data science is a collaborative activity. While you may talk with others about the project, we ask that you **write your solutions individually**. If you do discuss the assignments with others please **include their names** at the top of your notebook.

Collaborators: *list collaborators here*

1.3 This Assignment

In this project, you will be building and improving on the concepts and functions that you implemented in Project 2A to create your own classifier to distinguish spam emails from ham (non-spam) emails. We will evaluate your work based on your model's accuracy and your written responses in this notebook.

After this assignment, you should feel comfortable with the following:

- Using `sklearn` libraries to process data and fit models
- Validating the performance of your model and minimizing overfitting
- Generating and analyzing precision-recall curves

1.4 Warning

This is a **real world** dataset– the emails you are trying to classify are actual spam and legitimate emails. As a result, some of the spam emails may be in poor taste or be considered inappropriate. We think the benefit of working with realistic data outweighs these inappropriate emails, and wanted to give a warning at the beginning of the project so that you are made aware.

[2]:
```
# Run this cell to suppress all FutureWarnings
import warnings
warnings.filterwarnings("ignore", category=FutureWarning)
```

1.5 Score Breakdown

Question	Points
1	6
2a	4
2b	2
3	3
4	15
Total	30

1.6 Setup and Recap

Here we will provide a summary of Project 2A 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.

[3]:
```
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)
```

1.6.1 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 8348 labeled examples, and the unlabeled test set contains 1000 unlabeled examples.

Run the following cell to load in the data into DataFrames.

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 1000 unlabeled emails. You will predict labels for these emails and submit your predictions to the autograder for evaluation.

[4]:
```
import zipfile
with zipfile.ZipFile('spam_ham_data.zip') as item:
    item.extractall()
```

[5]:
```
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()
```

[5]:

	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.

[6]:
```
# 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
```

1.6.2 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 will need this **validation data** to assess the performance of your classifier once you are finished training.

As in Project 2A, we set the seed (random_state) to 42. **Do not modify this in the following questions, as our tests depend on this random seed.**

[7]:
```
# 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,␣
 ↪random_state = 42)

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

1.6.3 Feature Engineering

In order to train a logistic regression model, we need a numeric feature matrix X and a vector of corresponding binary labels y. To address this, in Project 2A, we implemented the function `words_in_texts`, which creates numeric features derived from the email text and uses those features for logistic regression.

For this project, we have provided you with an implemented version of `words_in_texts`. Remember that the function outputs a 2-dimensional NumPy array containing one row for each email text. The row should contain either a 0 or a 1 for each word in the list: 0 if the word doesn't appear in the text and 1 if the word does.

[8]:
```
def words_in_texts(words, texts):
    '''
    Args:
        words (list): words to find
        texts (Series): strings to search in
```

```
    Returns:
        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␣
↪words]).T
    return indicator_array
```

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

[9]:
```
words_in_texts(['hello', 'bye', 'world'], pd.Series(['hello', 'hello␣
↪worldhello']))
```

[9]:
```
array([[1, 0, 0],
       [1, 0, 1]])
```

1.6.4 EDA and Basic Classification

In Project 2A, 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 `scikit-learn`.

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

[10]:
```
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]
```

[10]:
```
(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]))
```

[11]:
```
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
```

1.6.5 Evaluating Classifiers

In our models, we are evaluating accuracy on the training set, which may provide a misleading accuracy measure. In Project 2A, 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.

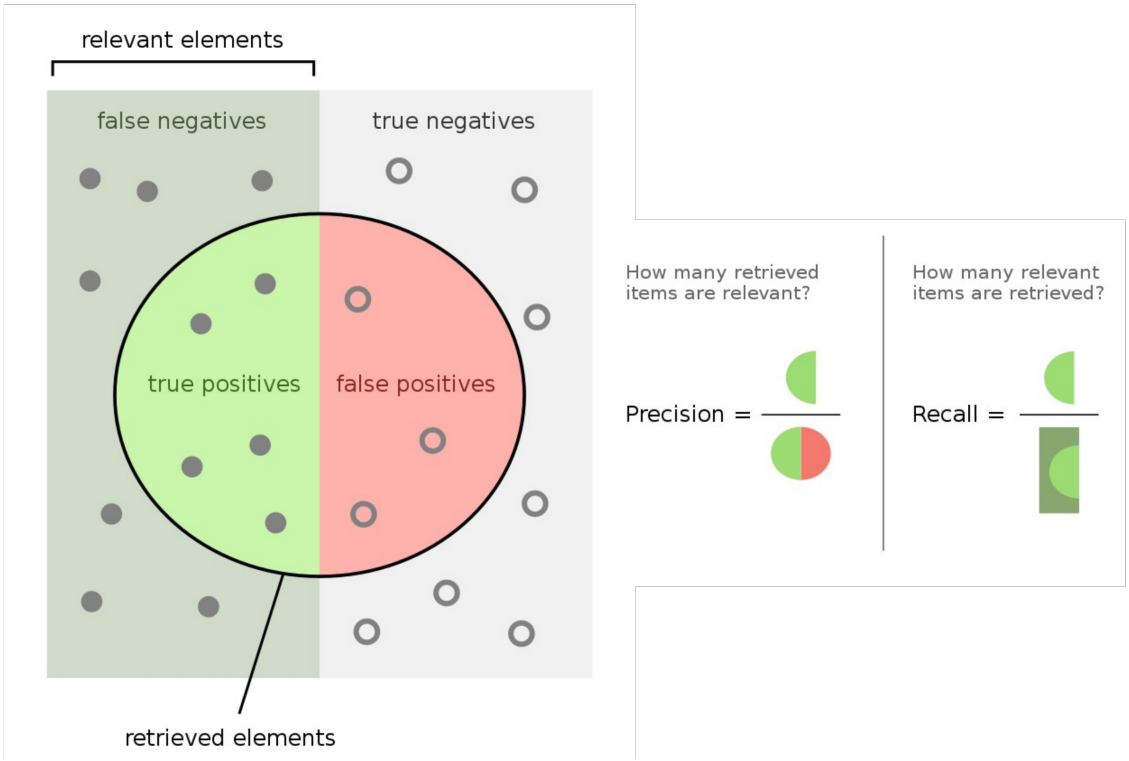
To be clear, we label spam emails as 1 and ham emails as 0. These definitions depend both on the true labels and the predicted labels. False positives and false negatives may be of differing importance, leading us to consider more ways of evaluating a classifier, in addition to overall accuracy:

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

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

False-alarm rate measures the proportion $\frac{\text{FP}}{\text{FP+TN}}$ of ham emails that were incorrectly flagged as spam.

The two graphics below 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.

2 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 **88%** accuracy on the test set. 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 four 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:
    1. Number of characters in the subject / body
    2. Number of words in the subject / body
    3. Use of punctuation (e.g., how many '!'s were there?)
    4. Number / percentage of capital letters
    5. 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 regularization parameter) 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.

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.* Note, 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.

---

[12]:
```
from collections import Counter
import re
```

[16]:
```
ham_emails = train[train["spam"] == 0]["email"].to_numpy()
spam_emails = train[train["spam"] == 1]["email"].to_numpy()

stop_words = ["about", "actually", "almost", "also", "although", "always",␣
 ↪"and", "any", "are",
              "became", "become", "but", "can", "could", "did", "does", "each",␣
 ↪"either", "else",
              "for", "from", "had", "has", "have", "hence", "how", "its",␣
 ↪"just", "maybe",  "might",
              "mine", "must", "neither", "nor", "not", "when", "where",␣
 ↪"whereas", "whenever",
              "whether", "which", "while", "who", "whom", "whoever", "whose",␣
 ↪"why", "with",
              "within", "without", "would", "yes", "yet", "you", "your"]

def ham_spam_words_count(ham_emails, spam_emails):
    pattern = r"[A-za-z]{3,}"
    ham_words = " ".join(ham_emails).replace("_", "").replace("[", "")
    ham_words = re.findall(pattern, ham_words)
    filtered_ham_words = [word for word in ham_words if word.casefold() not in␣
 ↪stop_words]
    counters_found_ham = Counter(filtered_ham_words)
    ham_words_count = counters_found_ham.most_common(1000)

    spam_words = " ".join(spam_emails).replace("_", "").replace("[", "")
```

```
    spam_words = re.findall(pattern, spam_words)
    filtered_spam_words = [word for word in spam_words if word.casefold() not␣
 ↪in stop_words]
    counters_found_spam = Counter(filtered_spam_words)
    spam_words_count = counters_found_spam.most_common(1000)
    return ham_words_count, spam_words_count

def design_matrix(df):
    ham_emails = train[train["spam"] == 0]["email"].to_numpy()
    spam_emails = train[train["spam"] == 1]["email"].to_numpy()

    ham_words_count, spam_words_count = ham_spam_words_count(ham_emails,␣
 ↪spam_emails)

    words_ham = [tup[0] for tup in ham_words_count]
    words_spam = [tup[0] for tup in spam_words_count]

    common_words = [x for x in words_ham if x in words_spam]

    train_words = [word for word in words_ham if word not in common_words]

    X_design = np.array(words_in_texts(train_words, df["email"]))
    return X_design
```

[17]:
```
# training data
X_train2 = design_matrix(train)
Y_train2 = np.array(train["spam"])
```

[18]:
```
# fit logistic regression model on training data
lrm = LogisticRegression(solver='lbfgs')
lrm.fit(X_train2, Y_train2)

training_acc = lrm.score(X_train2, Y_train2)
training_acc
```

[18]: 0.9696526021562625

[19]:
```
# val data
X_val = design_matrix(val)
Y_val = np.array(val["spam"])
val_acc = lrm.score(X_val, Y_val)
val_acc
```

[19]: 0.9568862275449102

2.0.1 Question 1: Feature/Model Selection Process

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?

1) To find better features, I had to do a lot of EDA on the given training data. For instance, I used regex to extract words that had at least 3 letters. I created a list of stop words to filter out stop words in order to find better words that would fit my model. I also used the Counter object to find the top 1000 words in both emails that were labeled as ham or spam. From the top 1000 words from both ham and spam, I basically took the conjunction of them to find the words that were very common in ham but not in spam.
2) I tried using the same model from proj2a using the same parameters (i.e. just solver='lbfgs'). Surprisingly, it worked very well with the feature engineering I developed to fit the model. Even without an intercept term, I was able to reach a high accuracy of at least 95% across the training data, validation data, and test data.
3) It took a lot of different tools to find good words (or features). For instance, I used regex to extract better words that I thought would fit the model well. I also used other tools we did not learn in class. For instance, I utilized the Counter class to find common words in a list to fine-tune these words into good features for my model.

Optional: Build a Decision Tree model with reasonably good accuracy. What features does the decision tree use first?

2.0.2 Question 2: EDA

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 2A. Specifically, don't show us a bar chart of proportions, or a one-dimensional class-conditional density plot. Any other plot is acceptable, **as long as it comes with thoughtful commentary.** Here are some ideas:

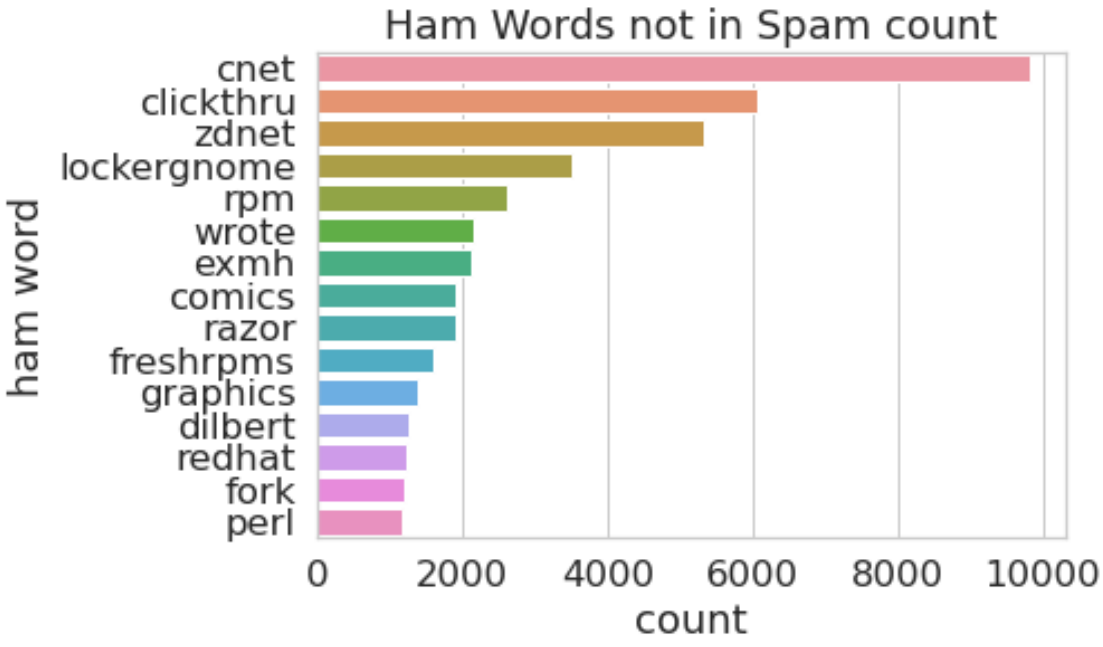
1. Consider the correlation between multiple features (look up correlation plots and `sns.heatmap`).
2. Try to show redundancy in a group of features (e.g. `body` and `html` might co-occur relatively frequently, or you might be able to design a feature that captures all html tags and compare it to these).

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.

[20]:
```
ham_words_count, spam_words_count = ham_spam_words_count(ham_emails,␣
 ↪spam_emails)

words_ham = [tup[0] for tup in ham_words_count]
words_spam = [tup[0] for tup in spam_words_count]

common_words = [x for x in words_ham if x in words_spam]
train_words = [word for word in words_ham if word not in common_words]

ham_words_not_in_spam = []
ham_words_not_in_spam_count = []
for tup in ham_words_count:
    if tup[0] in train_words:
        ham_words_not_in_spam.append(tup[0])
        ham_words_not_in_spam_count.append(tup[1])

ham_words_df = pd.DataFrame(data={"ham word": ham_words_not_in_spam, "count":␣
 ↪ham_words_not_in_spam_count})
top_15_ham_words_df = ham_words_df.iloc[0:15, :]
ax = sns.barplot(data=top_15_ham_words_df, x="count", y="ham word")
plt.title("Ham Words not in Spam count")
```

[20]:
```
Text(0.5, 1.0, 'Ham Words not in Spam count')
```



Question 2b Write your commentary in the cell below.

The plot above shows the top 15 words in emails labeled as ham in the training data. These words are selected to fit the model as best as possible in order to make good predictions and achieve high accuracy. For instance, the words must: 1) be at least 3 letters long, 2) only contain lowercase letters (no numbers, special characters, etc.), 3) also not be in spam emails at all. These are just 15 words, but I used almost 500 words as features to train the model.

2.0.3 Question 3: ROC Curve

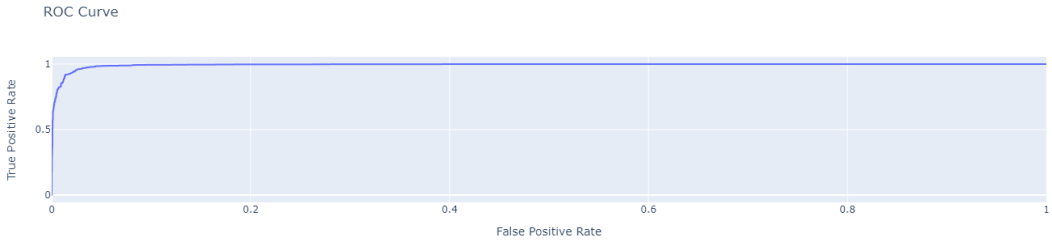
In most cases we won't be able to get 0 false positives and 0 false negatives, so we have to compromise. For example, in the case of cancer screenings, false negatives are comparatively worse than false positives — a false negative means that a patient might not discover that they have cancer until it's 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*: 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 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 20 to see how to plot an ROC curve.

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

[21]:
```
from sklearn.metrics import roc_curve
import plotly.express as px

scores = lrm.predict_proba(X_train2)
fprs, tprs, thresholds = roc_curve(Y_train, scores[:, 1])
fig = px.line(x=fprs, y = tprs, hover_name=thresholds, title="ROC Curve")
fig.update_xaxes(title="False Positive Rate")
fig.update_yaxes(title="True Positive Rate")
fig.show()
```



3 Question 4: 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 2B Test Predictions" assignment on Gradescope to get credit for this question.**

Save your predictions in 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.

Note: You may submit up to 4 times a day. If you have submitted 4 times on a day, you will need to wait until the next day for more submissions.

Note that this question is graded on an absolute scale based on the accuracy your model achieves on the overall test set, and as such, your score does not depend on your ranking on Gradescope.

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

[22]:
```
test_predictions = lrm.predict(design_matrix(test))
```

[23]:
```
grader.check("q4")
```

[23]:
```
q4 results: All test cases passed!
```

The following cell generates a CSV file with your predictions. **You must submit this CSV file to the "Project 2B Test Predictions" assignment on Gradescope to get credit for this question.** There are a maximum of 4 attempts per day of submitting to this assignment, so please use them wisely!

Note that the file will appear in your DataHub, you must navigate to the `proj2b` directory in your DataHub to download the file.

[24]:
```
from datetime import datetime

# Assuming that your predictions on the test set are stored in a 1-dimensional␣
 ↪array called
# test_predictions. Feel free to modify this cell as long you create a CSV in␣
 ↪the right format.

# Construct and save the submission:
```

```
submission_df = pd.DataFrame({
    "Id": test['id'],
    "Class": test_predictions,
}, columns=['Id', 'Class'])
timestamp = datetime.isoformat(datetime.now()).split(".")[0]
submission_df.to_csv("submission_{}.csv".format(timestamp), index=False)

print('Created a CSV file: {}.'.format("submission_{}.csv".format(timestamp)))
print('You may now upload this CSV file to Gradescope for scoring.')
```

```
Created a CSV file: submission_2022-08-08T20:13:56.csv.
You may now upload this CSV file to Gradescope for scoring.
```

3.1 Congratulations! You have completed Project 2B!

To double-check your work, the cell below will rerun all of the autograder tests.

[27]:
```
grader.check_all()
```

[27]:
```
q4 results: All test cases passed!
```

3.2 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!**

[28]:
```
# Save your notebook first, then run this cell to export your submission.
grader.export()
```

```
/opt/conda/lib/python3.9/site-packages/nbconvert/filters/datatypefilter.py:41:
UserWarning:

Your element with mimetype(s) dict_keys(['text/html']) is not able to be
represented.


<IPython.core.display.HTML object>
```