Lecture 6: Embeddings and Model Selection

Anushna Prakash
Data 598 (Winter 2022), University of Washington

This is my submission for Assignment 6. The Bonus part has been filled in towards the bottom of the notebook.

We will discuss two topics this lecture:

- Embeddings for natural language
- Model Selection with statistical tests

The first part of this notebook has been adapted from the D2L book.

Part 1: Embeddings for Natural Language

The field of **natural language processing (NLP)** is concerned with the interaction between computers and natural (human) language. This involves "understanding" the contents of documents, including the contextual nuances of the language within them.

Embeddings: The use of machine learning for NLP, both in the classical settings as well as the modern deep learning era, have relied on *embedding* words in vector spaces. Words are made of characters, which are combinatorial in nature with no "neighborhood" structure which one expects of vectors in, say, a Euclidean space. The magic of embeddings is that they are able to capture some "neighborhood" structure in words, e.g., the embedding of synonyms are closer together than of words which have nothing in common.

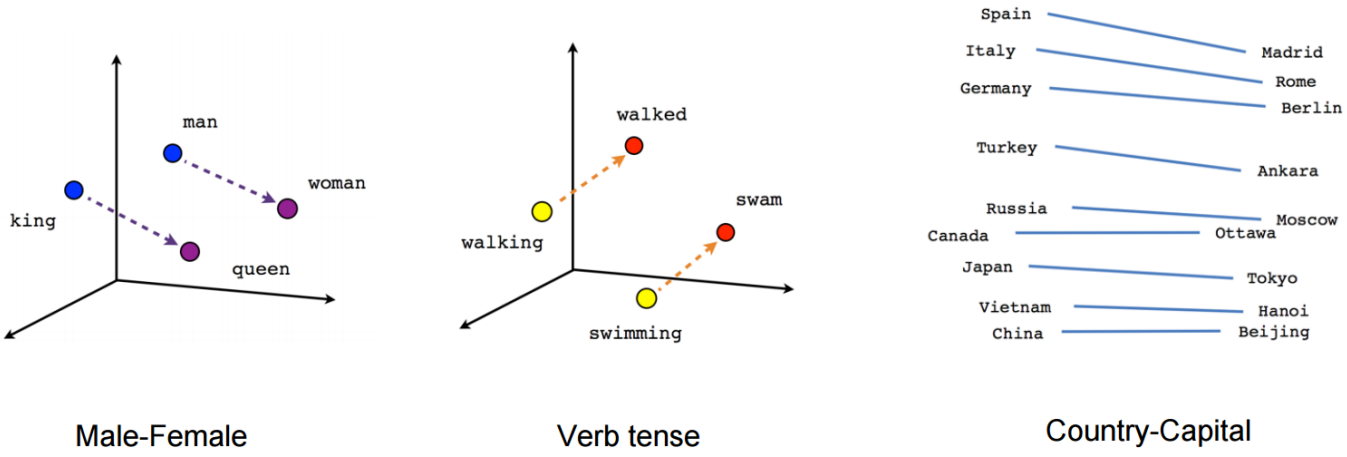

Image credits: https://towardsdatascience.com/creating-word-embeddings-coding-the-word2vec-algorithm-in-python-using-deep-learning-b337d0ba17a8

Note: Sometimes, we will work at the level of subword units, rather than words. Mathematically, the same treatment holds irrespective of how we *tokenize* the text. We will refer to these units as *tokens*.

Types of embeddings:

- Global embeddings: word2vec, GloVe
- Contextual embeddings: ELMo, BERT, ...

In this lab, we will play with the GloVe embeddings, which are global embeddings.

In [1]:
```
import numpy as np
```

In [2]:
```
# Download the GloVe embedding ~66M compressed + 164M uncompressed
import os
if 'glove.6B.50d' not in os.listdir():
```

```
    !wget http://d2l-data.s3-accelerate.amazonaws.com/glove.6B.50d.zip
    !unzip glove.6B.50d.zip
```

```
--2022-03-01 20:56:34--  http://d2l-data.s3-accelerate.amazonaws.com/glove.6B.50d.zip
Resolving d2l-data.s3-accelerate.amazonaws.com (d2l-data.s3-accelerate.amazonaws.com)... 52.84.162.136
Connecting to d2l-data.s3-accelerate.amazonaws.com (d2l-data.s3-accelerate.amazonaws.com)|52.84.162.136|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 69182829 (66M) [application/zip]
Saving to: ‘glove.6B.50d.zip’

glove.6B.50d.zip    100%[===================>]  65.98M  9.75MB/s    in 6.8s

2022-03-01 20:56:41 (9.64 MB/s) - ‘glove.6B.50d.zip’ saved [69182829/69182829]

Archive:  glove.6B.50d.zip
   creating: glove.6B.50d/
  inflating: glove.6B.50d/vec.txt
```

We index each word in our dictionary using integers. We store the following mappings:

- word $\rightarrow$ index
- index $\rightarrow$ word
- index $\rightarrow$ embedding of the corresponding word

Words not in our dictionary are denoted using the `<unk>` token

In [3]:
```
class GloVeWordEmbedding:
    def __init__(self):
        self.idx_to_token, self.idx_to_vec = self._load_embedding()
        self.unknown_idx = 0
        self.token_to_idx = {token: idx for idx, token in
                             enumerate(self.idx_to_token)}

    def _load_embedding(self):
        idx_to_token, idx_to_vec = ['<unk>'], []
        with open('glove.6B.50d/vec.txt') as f:
            for line in f:
                elems = line.rstrip().split(' ')
                token, elems = elems[0], [float(elem) for elem in elems[1:]]
                # Skip header information, such as the top row in fastText
                if len(elems) > 1:
                    idx_to_token.append(token)
                    idx_to_vec.append(elems)
        idx_to_vec = [[0] * len(idx_to_vec[0])] + idx_to_vec
        return idx_to_token, np.asarray(idx_to_vec)

    def __getitem__(self, tokens):
        # "tokens" is a list of words
        # use as object[tokens]
        # map token -> index -> vector
        indices = [self.token_to_idx.get(token, self.unknown_idx)
                   for token in tokens]
        vecs = self.idx_to_vec[np.asarray(indices)]
        return vecs

    def __call__(self, tokens):
        # Use as object(tokens)
        return self.__getitem__(tokens)

    def __len__(self):
        return len(self.idx_to_token)
```

In [4]:
```
glove_embedding = GloVeWordEmbedding()
```

We start by noting that the embedding of a word does not depend on its context.

In [5]:
```
# To obtain the embeddings of words:
sentence1 = 'I love data science'
embeddings1 = glove_embedding[sentence1.split()]  # using the __getitem__ method
# alternatively:
# embeddings1 = glove_embedding(sentence1.split())  # using the __call__ method
print(embeddings1.shape)  # (number of words, dimension)
```

```
(4, 50)
```

In [6]:
```
sentence2 = 'As a kid, I always wanted to study mathematics and science'
```

```
embeddings2 = glove_embedding[sentence2.split()]
print(embeddings2.shape)  # (number of words, dimension)
```

```
(11, 50)
```

In [7]:
```
# Compare the embeddings of both the time the word "science" appears
e1 = embeddings1[-1]
e2 = embeddings2[-1]
print(np.linalg.norm(e1-e2))
```

```
0.0
```

Next we will look at the cosine similarity between word embeddings. Recall that the cosine similarity between two vectors $u, v \in \mathbb{R}^d$ is defined as

$$S_{\cos}(u, v) = \frac{\langle u, v \rangle}{\|u\|_2 \, \|v\|_2}.$$

For any pair of vectors, the cosine similarity is always between -1 and 1. (Why?)

In [8]:
```
# Write a function to compute the k nearest neighbors
# as per cosine similarity
def k_nearest_neighbors(population, query, k):
    # population is a matrix of size (n, dim)
    # query is a matrix of shape (1, dim)
    # k is an integer
    # return topk indices and topk values
    cos = np.dot(population, query.reshape(-1,)) / (
        np.sqrt(np.sum(population * population, axis=1) + 1e-9) *
        np.linalg.norm(query.reshape(-1))
    )
    topk_idx = np.argpartition(cos, -k)[-k:] # unsorted
    topk_idx = topk_idx[np.argsort(cos[topk_idx])][::-1]  # sorted
    topk_val = cos[topk_idx]
    return topk_idx, topk_val

# Hint for topk in numpy: https://stackoverflow.com/a/23734295
```

In [9]:
```
query = glove_embedding(['microwave'])
topk_idxs, topk_vals = k_nearest_neighbors(glove_embedding.idx_to_vec, query, 6)
topk_words = [glove_embedding.idx_to_token[i] for i in topk_idxs]

for i, (w, val) in enumerate(zip(topk_words, topk_vals)):
    print(i, w, val)
```

```
0 microwave 0.9999999999839023
1 analog 0.7300107691175304
2 microwaves 0.7264979322621883
3 oven 0.7115686481570181
4 refrigerator 0.7039825402692077
5 ovens 0.6948281242683831
```

In [10]:
```
glove_embedding.idx_to_vec[0]
```

Out[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., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
       0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])
```

In [11]:
```
query = glove_embedding(['serendipity'])
topk_idxs, topk_vals = k_nearest_neighbors(glove_embedding.idx_to_vec, query, 6)
topk_words = [glove_embedding.idx_to_token[i] for i in topk_idxs]

for i, (w, val) in enumerate(zip(topk_words, topk_vals)):
    print(i, w, val)
```

```
0 serendipity 0.9999999999485274
1 profundity 0.7222786942557122
2 happenstance 0.7016944428733708
3 strangeness 0.6975966805924638
4 weirdness 0.6901399894427984
5 hokum 0.6882356254537078
```

Analogies with word embeddings

In addition to seeking synonyms, we can also use the pretrained word vector to seek the analogies between words. For example, “man”:“woman”::“son”:“daughter” is an example of analogy, “man” is to “woman” as “son” is to “daughter”.

The problem of seeking analogies can be defined as follows: for four words in the analogical relationship $a : b :: c : d$, given the first three words, a, b and c, we want to find d.

Assume the word vector for the word w is $\text{vec}(w)$. To solve the analogy problem, we need to find the word vector that is most similar to the result vector of $\text{vec}(c) + \text{vec}(b) - \text{vec}(a)$.

In [12]:
```
def get_analogy(token_a, token_b, token_c):
    # Implement the analogy from above
    vecs = glove_embedding[[token_a, token_b, token_c]]
    x = vecs[1] - vecs[0] + vecs[2]
    topk, cos = k_nearest_neighbors(glove_embedding.idx_to_vec, x, 1)
    return glove_embedding.idx_to_token[int(topk[0])]  # Remove unknown words
```

In [13]:
```
get_analogy('man', 'woman', 'son')
```

Out[13]:
```
'daughter'
```

In [14]:
```
# capital-country
get_analogy('london', 'england', 'paris')
```

Out[14]:
```
'france'
```

In [15]:
```
# tense
get_analogy('dance', 'danced', 'look')
```

Out[15]:
```
'looked'
```

Part 2: Model Selection with Statistical Tests

We will run a test to compare two different models. This is called the McNemar's test.

Let h_1 and h_2 be two different classification algorithms. The hypotheses we're testing are:

$$\begin{aligned} H_0 : &\quad \text{acc}(h_1) = \text{acc}(h_2) \\ H_1 : &\quad \text{acc}(h_1) \neq \text{acc}(h_2), \end{aligned}$$

where $\text{acc}(h)$ is the accuracy of the classifier h.

To distinguish between the two, we compute two the following numbers:

- N_{01}: the number of validation examples misclassified by h_1 but correctly classified by h_2
- N_{10}: the number of validation examples correctly classified by h_1 but misclassified by h_2.

The test statistic is

$$T = \frac{(|N_{01} - N_{10}| - 1)^2}{N_{10} + N_{01}}.$$

Its asymptotic distribution under the null is χ^2-distribution with 1 degree of freedom. We reject the test null hypothesis if

$$T > \chi^2_{1,\alpha}$$

the $(1 - \alpha)$-quantile of the χ^2_1 distribution.

The exercise: Run this hypothesis test to compare the MLP from week 1 and the ConvNet from week 2. Use a significance $\alpha = 0.01$. Train each network with SGD for 30 passes with an appropriate learning rate.

Data

In [16]:
```
import numpy as np
import torch
from torchvision.datasets import MNIST, FashionMNIST
from torch.nn.functional import cross_entropy
import time
import scipy.stats
```

In [17]:
```
# download dataset (~117M in size)
```

```
train_dataset = FashionMNIST('./data', train=True, download=True)
X_train = train_dataset.data # torch tensor of type uint8
y_train = train_dataset.targets # torch tensor of type Long
test_dataset = FashionMNIST('./data', train=False, download=True)
X_test = test_dataset.data
y_test = test_dataset.targets

# choose a subsample of 10% of the data:
idxs_train = torch.from_numpy(
    np.random.choice(X_train.shape[0], replace=False, size=X_train.shape[0]//10))
X_train, y_train = X_train[idxs_train], y_train[idxs_train]
# idxs_test = torch.from_numpy(
#     np.random.choice(X_test.shape[0], replace=False, size=X_test.shape[0]//10))
# X_test, y_test = X_test[idxs_test], y_test[idxs_test]

print(f'X_train.shape = {X_train.shape}')
print(f'n_train: {X_train.shape[0]}, n_test: {X_test.shape[0]}')
print(f'Image size: {X_train.shape[1:]}')

# Normalize dataset: pixel values lie between 0 and 255
# Normalize them so the pixelwise mean is zero and standard deviation is 1

X_train = X_train.float()  # convert to float32
X_train = X_train.view(-1, 784)
mean, std = X_train.mean(axis=0), X_train.std(axis=0)
X_train = (X_train - mean[None, :]) / (std[None, :] + 1e-6)  # avoid divide by zero

X_test = X_test.float()
X_test = X_test.view(-1, 784)
X_test = (X_test - mean[None, :]) / (std[None, :] + 1e-6)

n_class = np.unique(y_train).shape[0]
```

```
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-images-idx3-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-images-idx3-ubyte.gz to ./data/FashionMNIST/raw/trai
n-images-idx3-ubyte.gz
  0%|          | 0/26421880 [00:00<?, ?it/s]
Extracting ./data/FashionMNIST/raw/train-images-idx3-ubyte.gz to ./data/FashionMNIST/raw

Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-labels-idx1-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-labels-idx1-ubyte.gz to ./data/FashionMNIST/raw/trai
n-labels-idx1-ubyte.gz
  0%|          | 0/29515 [00:00<?, ?it/s]
Extracting ./data/FashionMNIST/raw/train-labels-idx1-ubyte.gz to ./data/FashionMNIST/raw

Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-images-idx3-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-images-idx3-ubyte.gz to ./data/FashionMNIST/raw/t10k-
images-idx3-ubyte.gz
  0%|          | 0/4422102 [00:00<?, ?it/s]
Extracting ./data/FashionMNIST/raw/t10k-images-idx3-ubyte.gz to ./data/FashionMNIST/raw

Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-labels-idx1-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-labels-idx1-ubyte.gz to ./data/FashionMNIST/raw/t10k-
labels-idx1-ubyte.gz
  0%|          | 0/5148 [00:00<?, ?it/s]
Extracting ./data/FashionMNIST/raw/t10k-labels-idx1-ubyte.gz to ./data/FashionMNIST/raw

X_train.shape = torch.Size([6000, 28, 28])
n_train: 6000, n_test: 10000
Image size: torch.Size([28, 28])
```

Modules and SGD

In [18]:

```
import torch
class MLP(torch.nn.Module):
    def __init__(self, hidden_width=32):
        super().__init__()
        self.linear1 = torch.nn.Linear(784, hidden_width)
        self.linear2 = torch.nn.Linear(32, 10)
    def forward(self, x):
        x = self.linear1(x)
        x = torch.nn.functional.relu(x)
        x = self.linear2(x)
        return x

class ConvNet(torch.nn.Module):
    def __init__(self,num_classes=10):
        super().__init__()
```

```
        self.conv_ensemble_1 = torch.nn.Sequential(
            torch.nn.Conv2d(1, 16, kernel_size=5, padding=2),
            torch.nn.ReLU(),
            torch.nn.MaxPool2d(2))
        self.conv_ensemble_2 = torch.nn.Sequential(
            torch.nn.Conv2d(16, 32, kernel_size=5, padding=2),
            torch.nn.ReLU(),
            torch.nn.MaxPool2d(2))
        self.fc = torch.nn.Linear(7*7*32, 10)

    def forward(self, x):
        x = x.view(-1, 1, 28, 28)
        out = self.conv_ensemble_1(x)
        out = self.conv_ensemble_2(out)
        out = out.view(out.shape[0], -1)
        out = self.fc(out)
        return out

# Some utility functions to compute the objective and the accuracy
def compute_objective(model, X, y):
    score = model(X)
    # PyTorch's function cross_entropy computes the multinomial logistic loss
    return cross_entropy(input=score, target=y, reduction='mean')

def sgd_one_pass(model, X, y, learning_rate, verbose=False):
    num_examples = X.shape[0]
    average_loss = 0.0
    for i in range(num_examples):
        idx = np.random.choice(X.shape[0])
        # compute the objective.
        # Note: This function requires X to be of shape (n,d). In this case, n=1
        objective = compute_objective(model, X[idx:idx+1], y[idx:idx+1])
        average_loss = 0.99 * average_loss + 0.01 * objective.item()
        if verbose and (i+1) % 100 == 0:
            print(average_loss)

        # compute the gradient using automatic differentiation
        gradients = torch.autograd.grad(outputs=objective, inputs=model.parameters())

        # perform SGD update. IMPORTANT: Make the update inplace!
        for (w, g) in zip(model.parameters(), gradients):
            w.data -= learning_rate * g.data


from tqdm.auto import trange # range + progress bar
def sgd_n_passes(model, X_train, y_train, X_val, y_val, n_passes, learning_rate):
    for i in trange(n_passes):
        sgd_one_pass(model, X_train, y_train, learning_rate)
    return compute_prediction_performance(model, X_val, y_val)

@torch.no_grad()
def compute_prediction_performance(model, X, y):
    # return a boolean vector of the same length as y
    # each entry is True if correctly predicted, else False
    pred = torch.argmax(model(X), axis=1)
    return y == pred
```

In [19]:
```
model1 = MLP()
performance1 = sgd_n_passes(
    model1, X_train, y_train, X_test, y_test, n_passes=30, learning_rate=2e-3
)
# boolean vector of length n_test
```

```
  0%|          | 0/30 [00:00<?, ?it/s]
```

In [20]:
```
model2 = ConvNet()
performance2 = sgd_n_passes(
    model2, X_train, y_train, X_test, y_test, n_passes=30, learning_rate=2.5e-3
)
# boolean vector of length n_test
```

```
  0%|          | 0/30 [00:00<?, ?it/s]
```

In [22]:
```
print('accuracy of MLP:', performance1.sum().item()/y_test.shape[0])
print('accuracy of ConvNet:', performance2.sum().item()/y_test.shape[0])
```

```
accuracy of MLP: 0.8325
accuracy of ConvNet: 0.8695
```

We will test whether this difference is statistically significant or not. From a first glance, it does appear to be statistically significant as the gap is quite large.

Compute N_{01} and N_{10} from the output of SGD above.

In [23]:
```
N01 = (~performance1 & performance2).sum().item()  # MLP is wrong but ConvNet is correct
N10 = (performance1 & ~performance2).sum().item()  # MLP is correct but ConvNet is wrong
```

Now compute the test statistic, the threshold, which is the $(1-\alpha)$ quantile of the χ^2_1 distribution and read off the conclusion of the test. Recall that we have $\alpha = 0.01$ here.

In [24]:
```
T = (abs(N01 - N10) - 1)**2 / (N01 + N10)  # statistic
threshold = scipy.stats.chi2(df=1).ppf(0.99)

print(f'Test statistic: {T}, threshold: {threshold}')

if T > threshold:
    print('Null rejected')
else:
    print('Failed to reject the null')
```

```
Test statistic: 118.60714285714286, threshold: 6.6348966010212145
Null rejected
```

Bonus Exercise: Sentiment Analysis using GloVe Embeddings

The goal of this exercise is to repeat perform sentiment analysis using the data from the demo, except using GloVe embeddings.

Download the data from here.

We will use movie reviews from Rotten Tomatoes. The sentiment labels are:

- 0 - negative
- 1 - somewhat negative
- 2 - neutral
- 3 - somewhat positive
- 4 - positive

Load and visualize data

In [28]:
```
import pandas as pd
filename = './train.tsv'
# keep one example per sentence (original data labels each phrase)
data = pd.read_csv(filename, sep='\t').groupby('SentenceId').first()
data = data.drop(columns=['PhraseId'])


data.head(4)
```

Out[28]:

SentenceId	Phrase	Sentiment
1	A series of escapades demonstrating the adage ...	1
2	This quiet , introspective and entertaining in...	4
3	Even fans of Ismail Merchant 's work , I suspe...	1
4	A positively thrilling combination of ethnogra...	3

Train-test split and featurize

In [29]:
```
data = data.sample(frac=1)  # shuffle
train_data = data[:7000]
test_data = data[7000:]
print(train_data.shape, test_data.shape)
```

```
(7000, 2) (1529, 2)
```

Let $\varphi(w)$ denote the embedding of word w. Recall that GloVe is a global embedding which does not depend on the context of the word.

For a piece of text denoted by words $T = (w_1, \cdots, w_n)$, we summarize it by the vector

$$\psi(T) := \frac{1}{n} \sum_{i=1}^{n} w_i$$

In [30]:
```
from tqdm.auto import tqdm

@torch.no_grad()
def featurize(x): # x is pd.Series with text
    features = []
    for sen in tqdm(x):
        sen = tokenizer.encode(sen, return_tensors='pt')
        outputs = model(sen, return_dict=True)
        embeddings = outputs.last_hidden_state.squeeze() # (len, dim)
        mean_embedding = embeddings.mean(axis=0)
        features.append(mean_embedding.numpy())
    return np.stack(features)  # (n, dim)
```

In [33]:
```
from transformers import BertTokenizer, BertModel

model_name = 'bert-base-uncased'
# Download the pre-trained model + tokenizer (a total of 440 MB)
tokenizer = BertTokenizer.from_pretrained(model_name) # to tokenize the text
model = BertModel.from_pretrained(model_name)  # PyTorch module

x_train = featurize(train_data['Phrase'])
y_train = train_data['Sentiment'].values

x_test = featurize(test_data['Phrase'])
y_test = test_data['Sentiment'].values
```

```
Some weights of the model checkpoint at bert-base-uncased were not used when initializing BertModel: ['cls.seq_relationship.weigh
t', 'cls.predictions.transform.LayerNorm.bias', 'cls.predictions.transform.dense.weight', 'cls.predictions.decoder.weight', 'cls.
seq_relationship.bias', 'cls.predictions.transform.dense.bias', 'cls.predictions.bias', 'cls.predictions.transform.LayerNorm.weig
ht']
- This IS expected if you are initializing BertModel from the checkpoint of a model trained on another task or with another archi
tecture (e.g. initializing a BertForSequenceClassification model from a BertForPreTraining model).
- This IS NOT expected if you are initializing BertModel from the checkpoint of a model that you expect to be exactly identical
(initializing a BertForSequenceClassification model from a BertForSequenceClassification model).
  0%|          | 0/7000 [00:00<?, ?it/s]
  0%|          | 0/1529 [00:00<?, ?it/s]
```

Train a simple logistic regression classifier to test performance

In [34]:
```
# We will reduce the data dimensionality
# This step is optional and only perform it if you
# find that it helps the test accuracy
from sklearn.decomposition import PCA
pca = PCA(n_components=0.99, random_state=1).fit(x_train)  # keep 99% of the explained variance
x_train = pca.transform(x_train)
x_test = pca.transform(x_test)
```

In [83]:
```
from sklearn.linear_model import LogisticRegression, LogisticRegressionCV
# TODO: Tune C with cross-validation
clf = LogisticRegressionCV(random_state=0, Cs=np.logspace(1e-4, 1e-1, 25), cv=5, max_iter=200).fit(x_train, y_train)
print(f'Best C after cross-validation: {clf.C_[0]}')
```

```
Best C after cross-validation: 1.02940790749159
```

In [84]:
```
y_train_pred = clf.predict(x_train)
y_test_pred = clf.predict(x_test)

print('Train accuracy:', (y_train_pred == y_train).mean())
print('Test accuracy:', (y_test_pred == y_test).mean())
```

```
Train accuracy: 0.6228571428571429
Test accuracy: 0.43361674296926095
```

In [104…
```
def test_sentence(sentence):
    sentence = pd.Series([sentence])
    sentence = featurize(sentence)
    sentence = pca.transform(sentence)
    print(f'The model predicts: {clf.predict(sentence)}')
```

In [106…

```
# Test random sentences
test_sentence('This movie is the best as sucking')
test_sentence('I think this is awesome')
test_sentence('The Room exceeds the concepts of good movie-making')
```

```
  0%|          | 0/1 [00:00<?, ?it/s]
The model predicts: [2]
  0%|          | 0/1 [00:00<?, ?it/s]
The model predicts: [4]
  0%|          | 0/1 [00:00<?, ?it/s]
The model predicts: [1]
```