CS 505 Homework 04: Classification

Due Friday 10/27 at midnight (1 minute after 11:59 pm) in Gradescope (with a grace period of 6 hours)

You may submit the homework up to 24 hours late (with the same grace period) for a penalty of 10%.

All homeworks will be scored with a maximum of 100 points; point values are given for individual problems, and if parts of problems do not have point values given, they will be counted equally toward the total for that problem.

Note: I strongly recommend you work in **Google Colab** (the free version) to complete homeworks in this class; in addition to (probably) being faster than your laptop, all the necessary libraries will already be available to you, and you don't have to hassle with `conda`, `pip`, etc. and resolving problems when the install doesn't work. But it is up to you! You should go through the necessary tutorials listed on the web site concerning Colab and stor [illegible] is always ready to help you resol [illegible]

I will post a "w [illegible]

Submission [illegible]

You must com [illegible] mitting the following two files in Gra [illegible]

- A file `HW0` [illegible] `nd Run All` before yo [illegible]
- A file `HW0` [illegible]

For best result [illegible] `-> Print Review` from [illegible] your browser and then `Save as` [illegible] dows machine -- just make sure it is readable and no cell contents have been cut off. Make it easy to grade!

The date and time of your submission is the last file you submitted, so if your IPYNB file is submitted on time, but your PDF is late, then your submission is late.

Collaborators (5 pts)

Describe briefly but precisely

1. Any persons you discussed this homework with and the nature of the discussion;
2. Any online resources you consulted and what information you got from those resources; and

3. Any AI agents (such as chatGPT or CoPilot) or other applications you used to complete the homework, and the nature of the help you received.

A few brief sentences is all that I am looking for here.

```
I learned about the process of word segmentation and model training from the documents of pytorch and spacy, and the usage methods of relevant machine learning models from the documents of sklearn.
```

In [15]:

```
import math
import numpy as np
from numpy.random import shuffle, seed, choice
from tqdm import tqdm
from collections import defaultdict, Counter
import pandas as pd
import re
import matplotlib.pyplot as plt

import torch
from torch.utils.data import Dataset,DataLoader
import torch
from torch.u                                               ader
from torchvi
from torch i

from torchvi
import torch

from sklearn
from sklearn
from sklearn                                               zer, CountVectorizer
```

Problem One: Exploring Shakespeare's Plays with PCA (45 pts)

In this problem, we will use Principal Components Analysis to look at Shakespeare's plays, as we discussed with a very different play/movie in lecture. Along the way, we shall use the tokenizer and the TF-IDF vectorizer from `sklearn,` a common machine learning library.

Note: There is a library for text analysis in Pytorch called `Torchtext`, however, in my view this will less well-developed and less well-supported than the rest of Pytorch, so we shall use `sklearn` for this problem.

Part A: Reading and exploring the data (5 pts)

The cells below read in three files and convert them to `numpy` arrays (I prefer to work with the arrays rather than with `pandas` functions, but it is your choice).

1. The file `shakespeare_plays.csv` contains lines from William Shakespeare's plays. The second column of the file contains the name of

the play, the third the name of the player (or the indication `<Stage Direction>`, and the fourth the line spoken.

2. The file p … logy of Shakepea … second the genre, and the th … st performed. The plays are i …

3. The file p … aracter (defined somewhat … 1400 characters) in the first c …

For each of the …

In [16]:
```
plays_array … der/cs505/shakespear
print(plays_
player_gende … u/fac/snyder/cs505/p
print(player … ])
play_attribu … du/fac/snyder/cs505/
print(play_a … [0])
```

```
(111582, 4) [1 'Henry IV Part 1' '<Stage Direction>' 'ACT I']
(398, 2) ['AARON' 'male']
(36, 3) ['Henry IV Part 1' 'History' 15]
```

Part B: Visualizing the Plays (8 pts)

1. Create an array containing 36 strings, each being the concatenation of all lines spoken. Be sure to NOT include stage directions! You may wish to create an appropriate dictionary as an intermediate step.

2. Create a document-term matrix where each row represents a play and each column represents a term used in that play. Each entry in this matrix represents the number of times a particular word (defined by the column) occurs in a particular play (defined by the row). Use `CountVectorizer` in `sklearn` to create the matrix.

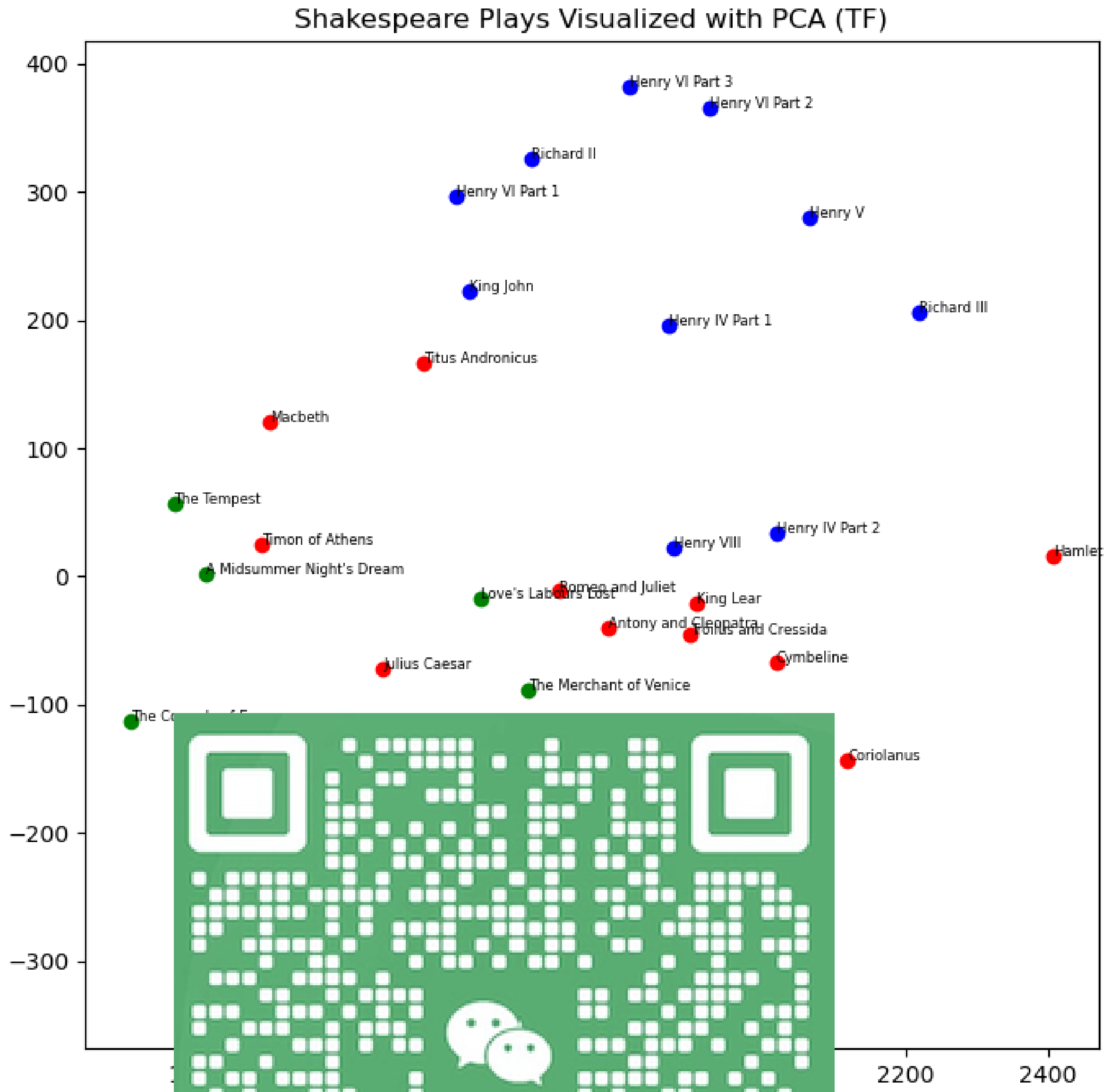
Keep the rows in the same order as in the original files in order to associate play names with terms correctly.

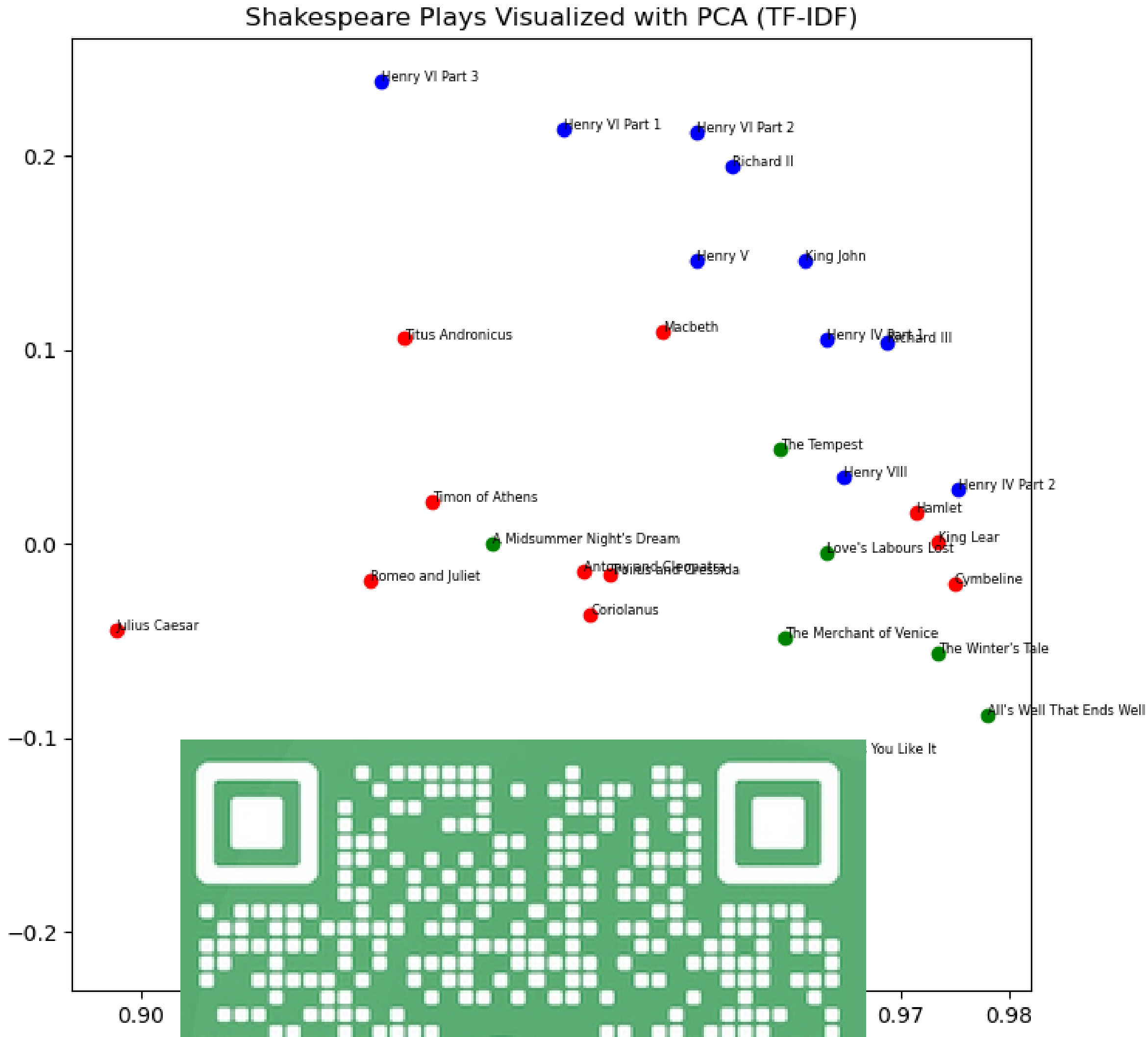
3. From this matrix, use `TruncatedSVD` in `sklearn` to create a 2-dimensional representation of each play. Try to make it as similar as possible to the illustration below, including (i) appropriate title, (ii) names of each play, followed by its chronological order, and (iii) different colors for each genre. Use a `figsize` of `(8,8)` and a `fontsize` of `6` to provide the best visibility. You can follow the tutorial here to create the visualization (look at the "PCA" part).

4. Now do the same thing all over again, but with TF-IDF counts (using `TFIDFVectorizer` in `sklearn`).

5. Answer the following in a few sentences: What plays are similar to each other? Do they match the grouping of Shakespeare's plays into comedies, histories, and tragedies here? Which plays are outliers (separated from the others in the same genre)? Did one of TF or TF-IDF provided the best insights?

In [94]:
```
genres_to_colors = {
    "History[illegible]
    "Comedy"[illegible]
    "Tragedy[illegible]
}

def visualiz[illegible]
    plt.figu[illegible]
    plt.titl[illegible]
    for i, ([illegible]_array):
        plt.[illegible]res_to_colors[genre
        plt.[illegible]tdict={"fontsize": 6
    plt.show[illegible]

plays_to_lin[illegible]
for _, play,[illegible]
    if playe[illegible]
        cont[illegible]
    plays_to[illegible]

strings = [][illegible]
for play, _, _ [illegible]
    strings.append(" ".join(plays_to_lines[play]))

svd = TruncatedSVD()

cv = CountVectorizer()
doc_term_mat = cv.fit_transform(strings)
reduced = svd.fit_transform(doc_term_mat)
visualize_pca_plays(reduced, "Shakespeare Plays Visualized with PCA (TF)")

tfidf = TfidfVectorizer()
doc_term_mat = tfidf.fit_transform(strings)
reduced = svd.fit_transform(doc_term_mat)
visualize_pca_plays(reduced, "Shakespeare Plays Visualized with PCA (TF-IDF)
```


Shakespeare Plays Visualized with PCA (TF)
400
300
200
100
0
−100
−200
−300
2200
2400
Henry VI Part 3
Henry VI Part 2
Richard II
Henry VI Part 1
Henry V
King John
Richard III
Henry IV Part 1
Titus Andronicus
Macbeth
The Tempest
Timon of Athens
Henry IV Part 2
Henry VIII
Hamlet
A Midsummer Night's Dream
Romeo and Juliet
King Lear
Antony and Cleopatra
Cymbeline
Julius Caesar
The Merchant of Venice
Coriolanus



Plays wi … ding to the
results … s with the
same ger … on the two-
dimensic … 's Dream and
Love's L … ided better
insights … s of PCA,
samples …

Part C: Visualizing the Players (8 pts)

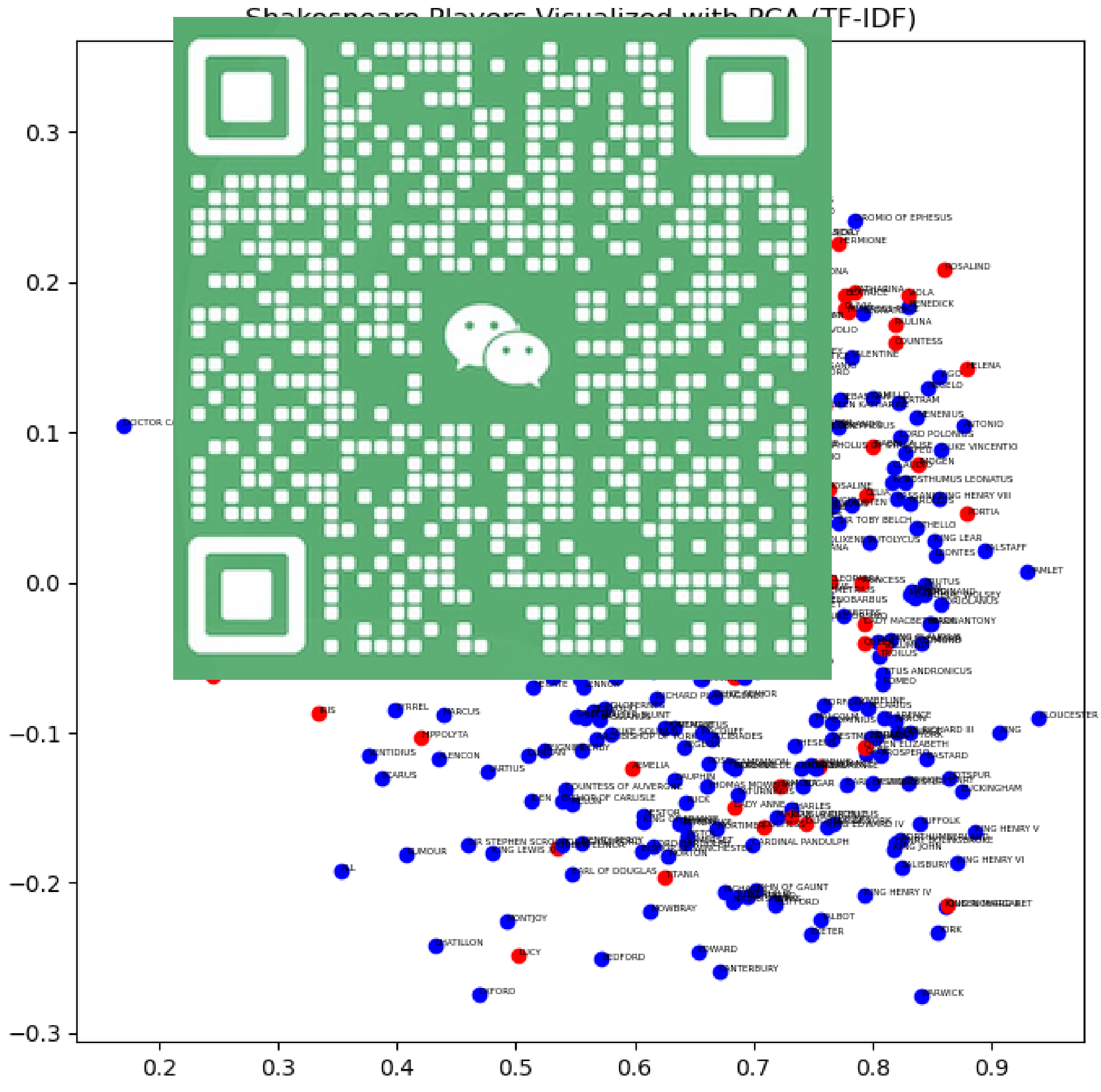
Now you must repeat this same kind of visualization, but instead of visualizing plays, you must visualize players. The process will be essentially the same, starting with an array of strings representing the lines spoken by each player. Use one of TF or TF-IDF, and use different colors for the genders.

Use a figsize of (8,8) and a fontsize of 4 to make this a bit more visible.

Again, comment on what you observe (it will not be as satisfying as the previous part).

In [96]:
```python
genders_to_colors = {
    "male": "blue",
    "female": "red",
}
```

```python
def visualize_pca_players(reduced, title):
    plt.figure(figsize=(8, 8))
    plt.title(title)
    for i, (player, gender) in enumerate(player_genders_array):
        plt.plot(reduced[i][0], reduced[i][1], 'o', c=genders_to_colors[genc
        plt.text(reduced[i][0], reduced[i][1], player, fontdict={"fontsize":
    plt.show()

players_to_lines = defaultdict(list)
for _, _, player, line in plays_array:
    if player == "<Stage Direction>":
        continue
    players_to_lines[player].append(line)

strings = []
for player, _ in player_genders_array:
    strings.append(" ".join(players_to_lines[player]))

doc_term_mat = tfidf.fit_transform(strings)
reduced = svd.fit_transform(doc_term_mat)
visualize_pca_players(reduced, "Shakespeare Players Visualized with PCA (TF-
```



From the visualization results of PCA, it can be seen that the red(female) and blue(male) samples are mixed together and

```
cannot be distinguished. So there is not much difference in
lines between roles of different genders.
```

Part D: DIY Word Embeddings (8 pts)

In this part you will create a word-word matrix where each row (and each column) represents a word in the vocabulary. Each entry in this matrix represents the number of times a particular word (defined by the row) co-occurs with another word (defined by the column) in a sentence (i.e., line in `plays`). Using the row word vectors, create a document-term matrix which represents a play as the average of all the word vectors in the play.

Display the plays using `TruncatedSVD` as you did previously.

Again, comment on what you observe: how different is this from the first visualization?

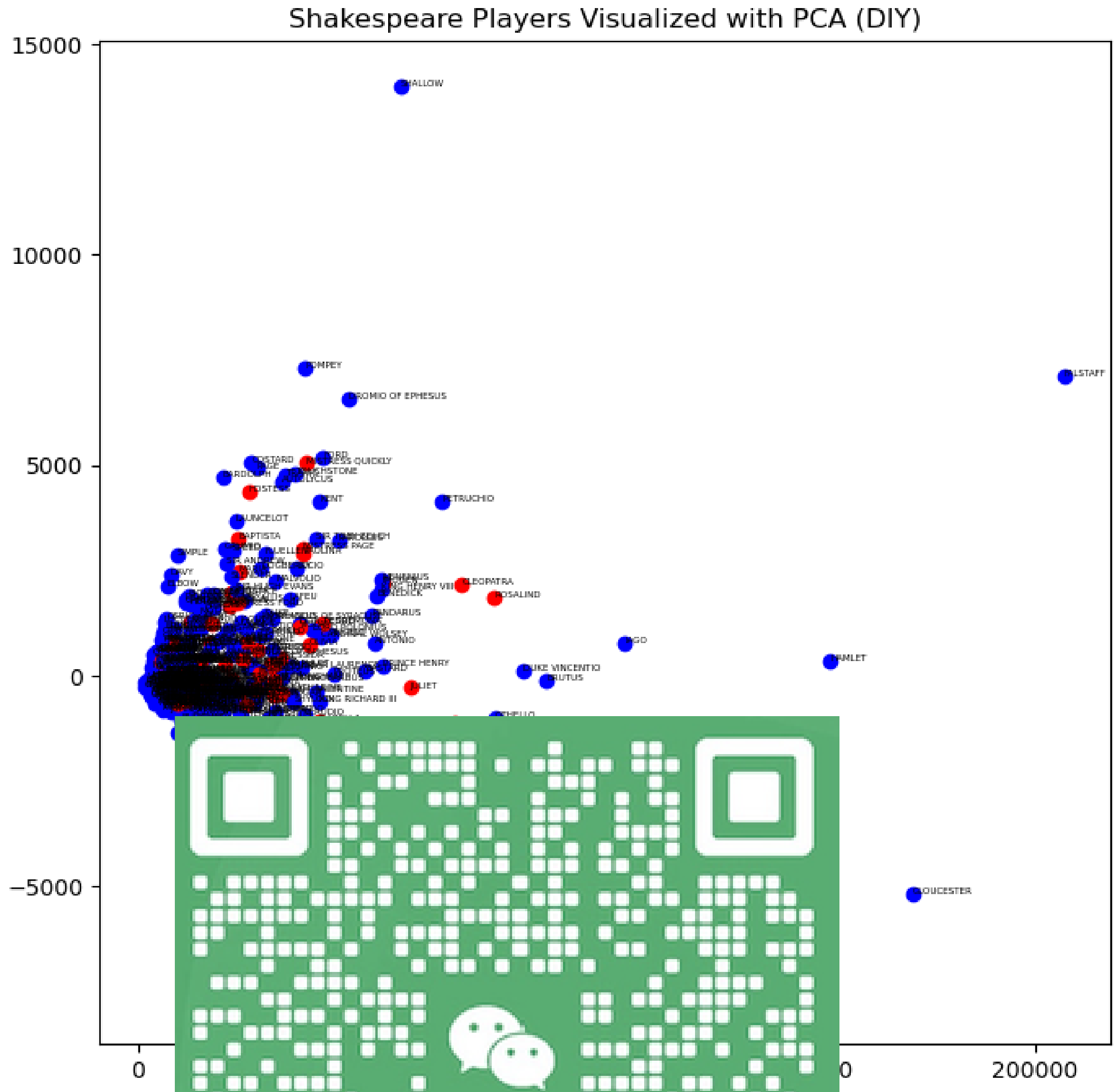
Notes:

1. Remove punctuation marks . , ; : ? ! but leave single quotes.
2. One way t ... ch word to a dictionary ... then from this to create the ... ment-term matrix which is in...
3. If you have ... ay wish to eliminate "stop wor... emaining most common w... e complete vocabular...

In [ ]:

```
import nltk
nltk.downloa

from nltk.co
stops = set(

dictionary =
for _, _, pl
    if playe
        continue
    words = line.split(" ")
    words = list(filter(None, map(lambda x: re.sub("[.,;:?!\t\n\"]", "", x).
    for word in words:
        dictionary[word] += 1

my_stops = set()

for word, count in dictionary.items():
    if count > 3000:
        my_stops.add(word)

stops = stops | my_stops

vocab = set(dictionary.keys()) - stops
N = len(vocab)
```

In [97]:
```
words_to_idx = {}
all_lines = []
players_to_words = defaultdict(lambda: defaultdict(int))

for _, _, player, line in plays_array:
    if player == "<Stage Direction>":
        continue
    words = line.split(" ")
    words = list(filter(None, map(lambda x: re.sub("[.,;:?!\t\n\"]", "", x).
    all_lines.append(words)
    for word in words:
        if word not in vocab:
            continue
        players_to_words[player][word] += 1

for i, word in enumerate(vocab):
    words_to_idx[word] = i

word_word_mat = np.zeros((N, N))

for line in all_lines:
    n = len(line)
    for i in
        word
        if w

        for



                                              _idx[word_j]] += 1


doc_term_mat
for player,
    vec = np
    words_to
    n = 0
    for word
        vec
        n +=
    vec /= n
    doc_term
doc_term_mat = np.array(doc_term_mat)

reduced = svd.fit_transform(doc_term_mat)
visualize_pca_players(reduced, "Shakespeare Players Visualized with PCA (DIY
```




```
Compared ... ample points
are more ... e, and there
are seve ... from other
samples.
```

Part E: Visualizing the Plays using Word2Vec Word Embeddings (8 pts)

Now we will do the play visualization using word embeddings created by Gensim's `Word2Vec`, which can create word embeddings just as you did in the previous part, but using better algorithms.

You can read about how to use `Word2Vec` and get template code here:

https://radimrehurek.com/gensim/models/word2vec.html

I strongly recommend you follow the directions for creating the model, then using `KeyedVectors` to avoid recomputing the model each time.

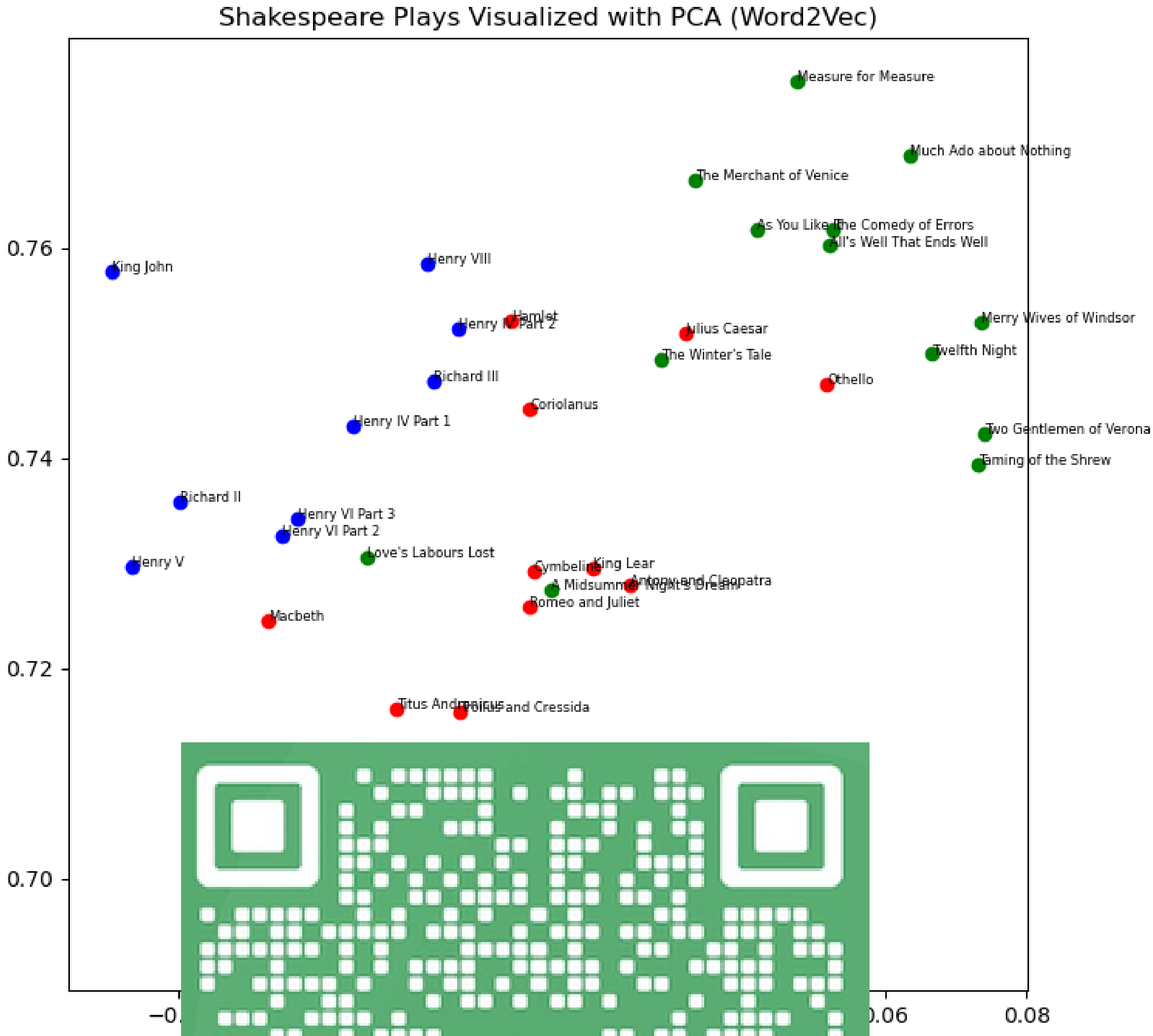
Experiment with the `window` (say 5) and the `min_count` (try in the range 1 - 5) parameters to get the best results.

Display the plays using `PCA` instead of `TruncatedSVD`.

Again, comment on what you observe: how different is this from the other visualizations?

In [91]:
```
from gensim.models import Word2Vec

model = Word2Vec(sentences=all_lines, vector_size=100, window=5, min_count=1
word_vectors = model.wv
word_vectors.save("word2vec.wordvectors")
```

In [95]:
```
from gensim.models import KeyedVectors
wv = KeyedVectors.load("word2vec.wordvectors", mmap='r')

plays_to_words = defaultdict(list)

for _, play, player, line in plays_array:
    if player == "<Stage Direction>":
        continue
    words = line.split(" ")
    words = list(filter(None, map(lambda x: re.sub("[.,;:?!\t\n\"]", "", x).
    plays_to_words[play] += words

doc_term_mat
for play, _,
    doc_term                                    bda x: wv[x], plays_
doc_term_mat

pca = PCA()
pca.fit_tran

visualize_pc                                    ualized with PCA (Wo
```



Unlike t A results,
blue sam d green
sample p

Part F: Vis ord Embeddin

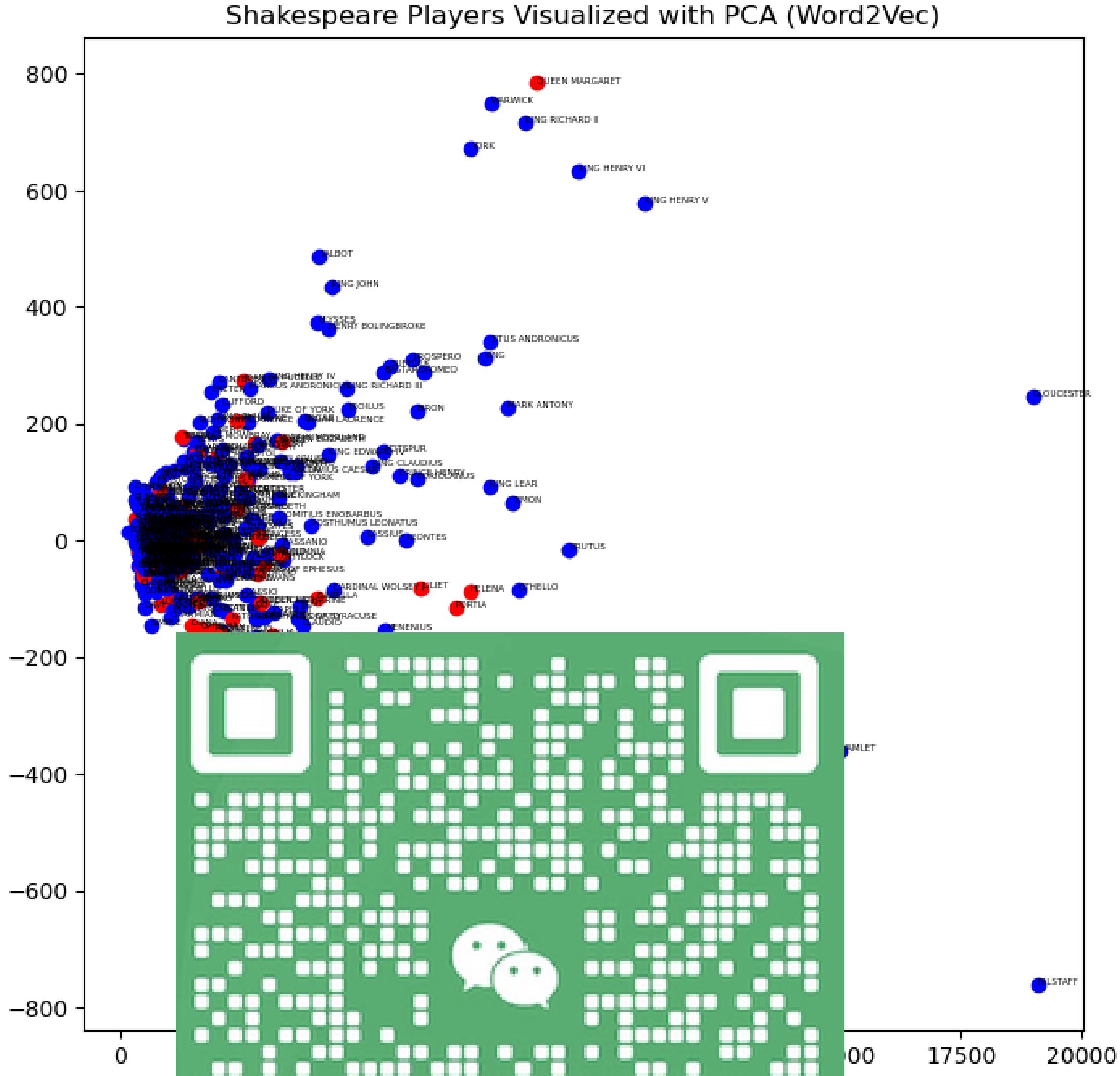
Now you must gs.

Use a figsize of (8,8) and a fontsize of 4 to make this a bit more visible.

Again, comment on what you observe. How is this different from what you saw in Part C?

In [99]:
```
doc_term_mat = []
for player, _ in player_genders_array:
    vec = 0
    words_to_nums = players_to_words[player]
    n = 0
    for word, number in words_to_nums.items():
        vec = vec + wv[word] * number
        n += number
    vec /= number
    doc_term_mat.append(vec)
doc_term_mat = np.array(doc_term_mat)
```

```
reduced = svd.fit_transform(doc_term_mat)
visualize_pca_players(reduced, "Shakespeare Players Visualized with PCA (Wor
```



The PCA [illegible] ent with DIY's.

Problem Two: Classifying Text with a Feed-Forward Neural Network (50 pts)

In this problem, you must create a FFNN in Pytorch to classify emails from the Enron dataset as to whether they are spam or not spam ("ham"). For this problem, we will use `Glove` pretrained embeddings. The dataset and the embeddings are in the following location:

https://drive.google.com/drive/folders/1cHR4VJuuN2tEpSkT3bOaGkOJrvIV-ISR?usp=sharing

(You can also download the embeddings yourself from the web; but the dataset is one created just for this problem.)

Part A: Prepare the Data (10 pts)

Compute the features of the emails (the vector of 100 floats input to the NN) vector based on the average value of the word vectors that belong to the words in it.

Just like the previous problem, we compute the 'representation' of each message, i.e. the vector, by averaging word vectors; but this time, we are using Glove word embeddings instead. Specifically, we are using word embedding 'glove.6B.100d' to obtain word vectors of each message, as long as the word is in the 'glove.6B.100d' embedding space.

Here are the steps to follow:

1. Have a basic idea of how Glove provides pre-trained word embeddings (vectors).
2. Download and extract word vectors from 'glove.6B.100d'.
3. Tokenize the messages (`spacy` is a good choice) and compute the message vectors by averaging the vectors of words in the message. You will need to test if a word is in the model (e.g., something like `if str(word) in glove_model ...`) and ignore any words which have no embeddings.

Part B: Cre

Now you must ... ting sets, and build a 'Dataset' an... model with Pytorch.

Use a train-val... nt with different batch sizes, st...

Hints:

1. Make sure ... our defined dataset are impler... uld return the specified ...
2. Don't com... m__ function, otherwise ... hese in an array before cre...
3. The data in the `.csv` is randomized, so you don't need to shuffle when doing the split.

In [ ]:
```
import spacy
nlp = spacy.load("en_core_web_sm")

embeddings = {}
with open("./glove.6B/glove.6B.100d.txt", "r") as f:
    lines = f.readlines()
    for line in lines:
        word_embedding = line.split(" ")
        word = word_embedding[0]
        embedding = np.array(list(map(lambda x: float(x), word_embedding[1:]
        embeddings[word] = embedding

class MyDataset(Dataset):
    def __init__(self, file_path) -> None:
```

```
        df = pd.read_csv(file_path)
        self._raw_data = []
        n = len(df)
        for i in range(n):
            message = df.iloc[i]["Message"]
            message = nlp(message)
            word_vecs = []
            for word in message:
                if word.text in embeddings.keys():
                    word_vecs.append(embeddings[word.text])
            vec = np.array(word_vecs)
            vec = np.average(vec, axis=0)
            self._raw_data.append((vec, df.iloc[i]["Spam"]))

    def __len__(self) -> int:
        return len(self._raw_data)

    def __getitem__(self, index) -> tuple[str, int]:
        return self._raw_data[index]

dataset = MyDataset("./data_pa5/enron_spam_ham.csv")
```

In [123…

```
seed = 42
torch.manual
np.random.se
batch_size =
trainset, de                                          .1, 0.1], generator
train_datalo                                          _size)
dev_dataload                                          e)
test_dataloa                                          ize)
```

Part C: Bui

Once the data … network model.

The model doe … ould be:

1. linear laye
2. ReLU acti
3. linear laye

But feel free to test out other possible combinations of linear layers & activation function and whether they make significant difference to the model performance later.

In order to perform "early stopping," you must keep track of the best validation score as you go through the epochs, and save the best model generated so far; then use the model which existed when the validation score was at a minimum to do the testing. (This could also be the model which is deployed, although we won't worry about that.) Read about `torch.save(...)` and `torch.load(...)` to do this.

Experiment with different batch sizes and optimizers and learning rates to get the best validation score for the model you create with early stopping. (Try not to look *too hard* at the final accuracy!) Include your final performance charts (using `show_performance_curves`) when you submit.

Conclude with a brief analysis (a couple of sentences is fine) relating what experiments you did, and what choices of geometry, optimizer, learning rate, and batch size gave you the best results. It should not be hard to get well above 90% accuracy on the final test.

In [124…

```
class MyModel(nn.Module):
    def __init__(self) -> None:
        super().__init__()
        self.classifier = nn.Sequential(
            nn.Linear(100, 15, dtype=torch.float64),
            nn.ReLU(),
            nn.Linear(15, 2, dtype=torch.float64),
            nn.Softmax()
        )

    def forward(self, x):
        return self.classifier(x)

max_epoch = 100
model = MyModel()
optimizer = optim.AdamW(model.parameters(), lr=1e-3)
loss_fn = nn.CrossEntropyLoss()
last_acc = 0
early_stop =
for i in ran
    model.tr
    for batc
        opti
        x, y
        pred
        loss
        loss
        opti
    model.ev
    correct
    for batc
        x, y
        pred
        pred
        corr
    acc = co
    acc = ac
    if acc <
        if last_acc - acc < 0.0001:
            early_stop += 1
            if early_stop == 5:
                break
        else:
            early_stop = 0
    else:
        last_acc = acc
        torch.save(model, "best_model.pt")

model = torch.load("best_model.pt")
model.eval()
correct = 0
for batch in test_dataloader:
    x, y = batch
    pred = model(x)
    pred = pred[:, 0] < .5
    correct = correct + (pred == y).sum()
```

```
acc = correct / len(testset)
acc = acc.item()
print("Test set accuracy: ", acc)
```

```
Test set accuracy:  0.9562744498252869
```

batch size	learning rate	optimizer	acc
64	**1e-3**	**AdamW**	**95.63**
64	1e-4	AdamW	94.13
64	1e-5	AdamW	86.60
64	1e-3	AdamW	94.67
128	1e-4	AdamW	93.60
128	1e-5	AdamW	83.29
64	1e-3	SGD	77.11
64	1e-2	SGD	93.49
128	1e-3	SGD	70.14
128	1e-2	SGD	91.72

The expe[illegible]tions of batch si[illegible]aluate their impact o[illegible], were compare[illegible]rning rates ranging [illegible]f 95.63% was achieve[illegible]f 1e-3, and AdamW op[illegible]s and larger batch si[illegible]nsistently outperfo[illegible]