playground

March 14, 2022

1 PSTAT131 Final Project

Sean Gao, Aya Zeplovitch

1.1 Introduction

This project aims to predict the number of comments given the question of any r/Askreddit question. r/Askreddit is a forum where redditors (Reddit users) post questions (about anything really). We try to find the answer to the following question: "what kind of r/Askreddit questions garner the most interactions?" This is an interesting question because as someone that regularly browses Reddit, there definitely exists a pattern between the posts that become popular.

1.2 NLP

As our data is going to be text data, we will have to use NLP techniques to transform text into machine readable dataframes. The most common techniques used are word-embedding, and bag-of-words. Here's approximately how they work:

Suppose we have the following sentence: *"I like my big pancakes"*

We can make our list of vocabulary from this sentence: **'I', 'like', 'my', big', 'pancakes'**. Both techniques uses the one-hot (or multi-hot in the case of bag-of-words) encoding (i.e., the word 'like' would be represented by a vector **[0, 1, 0, 0, 0]**).

1.2.1 Word-Embedding

In the case of word-embedding, a $m \times n$ matrix will be created, where m is the number of words in the sentence, and n is the number of words in our total vocabulary. As such, our sentence will be represented as such:

	I	like	my	big	pancakes
1	1	0	0	0	0
2	0	1	0	0	0
3	0	0	1	0	0
4	0	0	0	1	0
5	0	0	0	0	1

1.2.2 Bag-of-Words

In the case of bag-of-words, a single array of length m will be created. The value at each index will be the number of occurrences of the word that index represents. With the same vocabulary, *'I like pancakes.'* would be represented as such:

I	like	my	big	pancakes
1	1	0	0	1

1.2.3 Word-Embedding vs Bag-of-Words

Word-embedding holds more information than bag-of-words. For one, one sentence is represented as an $m \times n$ matrix with word-embedding, whereas it is represented as a m size array with bag-of-words. Most importantly, word-embedding is able to represent the order of the words in the sentence. For example, take the phrases "yeah no" and "no yeah." To a fluent English speaker, they have opposite meanings: "yeah no" means "no," and "no yeah" means "yes." However, they would have the same representation with the bag-of-words technique, whereas word-embedding would be able to properly represent the difference between these phrases.

However, word-embedding is much more computationally expensive as each sentence would be n times bigger. It also requires a different type of algorithm to model as each sentence would have a different n, making the number of predictors variable. As such, common machine learning methods like linear regression, ensemble tree methods, and many others would not work with the data. This is one reason we have decided to go with the bag-of-words approach. Another reason is that due to the nature of what we're trying to predict, the order of the words in the question may not matter as much as which words actually appear in the question.

1.3 Importing Packages

All the imports will be grouped at the top, make sure they are all installed and up to date if you're trying to run this notebook. This was bulit on Python 3.10~.

[ ]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import nltk
import pickle
import os

from sklearn.feature_extraction.text import CountVectorizer
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import cross_val_score
from sklearn.decomposition import PCA
from sklearn.preprocessing import scale
from sklearn.model_selection import RepeatedKFold
from sklearn.linear_model import LassoCV
```

```
from sklearn.model_selection import GridSearchCV
from sklearn.linear_model import LinearRegression
from sklearn.linear_model import Lasso
from sklearn.linear_model import Ridge

from scipy import stats
from tqdm import tqdm
```

1.4 Notebook Parameters

Some parameters here to help run the notebook faster. Our dataset is huge so it helps to only use portions of it for the sake of memory/computing power.

We were actually unable to use the entire dataset (nearly 1 million rows) as we lacked the memory. The number of words in the vocabulary was very important as well both for predictability and computational power. The `N_VOCAB` value we chose was mainly as a result of limited computational power and memory.

`PCT_DATA`: the percentage of the entire dataset used for the rest of the notebook.
`N_VOCAB`: the number of words in the vocabulary

[ ]:
```
PCT_DATA = .1
N_VOCAB = 500
```

1.5 Helper Functions

Each function will have Docstrings included to describe its purpose and usage

[ ]:
```
def rm_stop(words):
    """
    Removes the stopwords from a list of words
    """

    new_words = []
    stop_words = nltk.corpus.stopwords.words('english')

    for w in words:
        if w not in stop_words:
            new_words.append(w)
    return new_words

def build_data(pct_data=PCT_DATA):
    """
    Uses data/finaldataset.csv to create a text dataset.
    Result dataset will contain a string with stop words removed.
    """
```

```
    raw_df = pd.read_csv('data/finaldataset.csv')
    raw_df = raw_df.iloc[:round(raw_df.shape[0] * pct_data)]

    # num_comments is self explanatory, title is the question (our predictor)
    df = raw_df[['num_comments', 'title']]
    del raw_df

    # setting all strings to lowercase and tokenizing them
    df.loc[:,'title'] = df['title'].str.lower()
    tokenizer = nltk.RegexpTokenizer(r'\b[(a-z)]+\b')
    df.loc[:,'tokenized'] = df['title'].apply(tokenizer.tokenize)

    # removing stop words and rejoining the words for CountVectorizer later
    df['tokenized'] = df['tokenized'].apply(rm_stop)
    df['text'] = df['tokenized'].apply(lambda x: ' '.join([word for word in x]))

    return df

def fit_n_vocab(df, n_vocab=N_VOCAB):
    """
    Fits the bag-of-words with N_VOCAB as the number of words in the vocabulary.
    ↪

    Returns:
    --------
        X : np.array()
            Number of observations x N_VOCAB bag-of-words matrix
        Y : np.array()
            Array containing the number of comments
        vocab : np.array()
            Array of strings of the vocab
        vectorizer: CountVectorizer()
            CountVectorizer object used to transform new data
    """

    vectorizer = CountVectorizer(max_features=n_vocab)
    vecfit = vectorizer.fit_transform(df['text'])
    X = vecfit.toarray()

    vocab = vectorizer.get_feature_names()
    Y = df['num_comments'].to_numpy()

    return X, Y, vocab, vectorizer

def pct_no_words(X):
```

```
    """Calculates the percent of observations with 0 words in the␣
 ↪bag-of-words"""
    return sum(np.sum(X, axis=1) > 0)/X.shape[0]

def pred_from_str(q, m, vec):
    """Uses any model m to predict the number of comments of string q"""
    tokenizer = nltk.RegexpTokenizer(r'\b[(a-z)]+\b')

    q = tokenizer.tokenize(q)
    q = rm_stop(q)

    j = lambda x: ' '.join([word for word in x])
    q = j(q)
    pred_X = vec.transform(pd.Series(q)).toarray()

    pred = m.predict(pred_X)

    return pred
```

1.6 Our Data

Our data is gathered from the third party Reddit API pushshift. The code for our data gathering can be found in our Github repo here. Since the data collection and processing could take hours, we will include the link to the dataset here. The file we are using is *finaldataset.csv*.
The API did not yield the correct number of upvotes, which is why we're using the number of comments as our target variable. The number of comments or upvotes both show the popularity of the post, which is why we deemed the number of comments to be a sufficient replacement for the number of upvotes.

If you're trying to run this notebook, be sure to have your *finaldataset.csv* in a folder named "data", such that the file path from this file would be ***data/finaldataset.csv***.

[ ]:
```
df = build_data()
X, Y, vocab, vectorizer = fit_n_vocab(df=df, n_vocab=N_VOCAB)
del df
```

```
C:\Users\Sean\AppData\Local\Temp\ipykernel_2928\1935101033.py:20: DtypeWarning:
Columns (7,8,61,62,66,67,70,71,72,73,74,75) have mixed types. Specify dtype
option on import or set low_memory=False.
  raw_df = pd.read_csv('data/finaldataset.csv')
C:\Users\Sean\AppData\Local\Programs\Python\Python310\lib\site-
packages\sklearn\utils\deprecation.py:87: FutureWarning: Function
get_feature_names is deprecated; get_feature_names is deprecated in 1.0 and will
be removed in 1.2. Please use get_feature_names_out instead.
  warnings.warn(msg, category=FutureWarning)
```

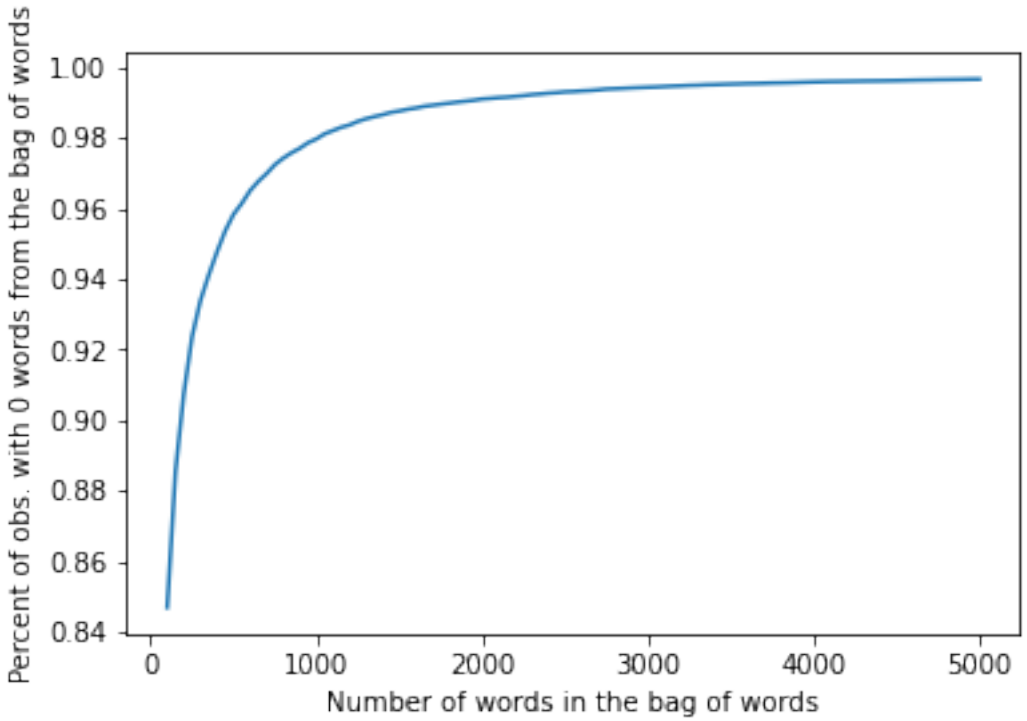
1.7 EDA

There are two main things we want to accomplish with EDA: the most practical `N_VOCAB` value and the best transformation for our target variable.

First we construct a graph plotting the number of vocabulary against the percentage of observations with at least 1 word in the bag-of-words (the horizontal sum of each observation is at least 1). This is important because having no word in the bag-of-words essentially means the data has no information regarding that observation. As such, we want to minimize the number of observations which the model learns nothing about. With our limited computational power, we are unable to have a large amount of vocabularies in our bag-of-words.

If you're running this from a cloned repo from our Github, you won't need to change anything to run this next code block, as *pct_df.csv* is included in the repo. Otherwise, set `run` to **True** if it is your first time running this notebook, **False** otherwise.

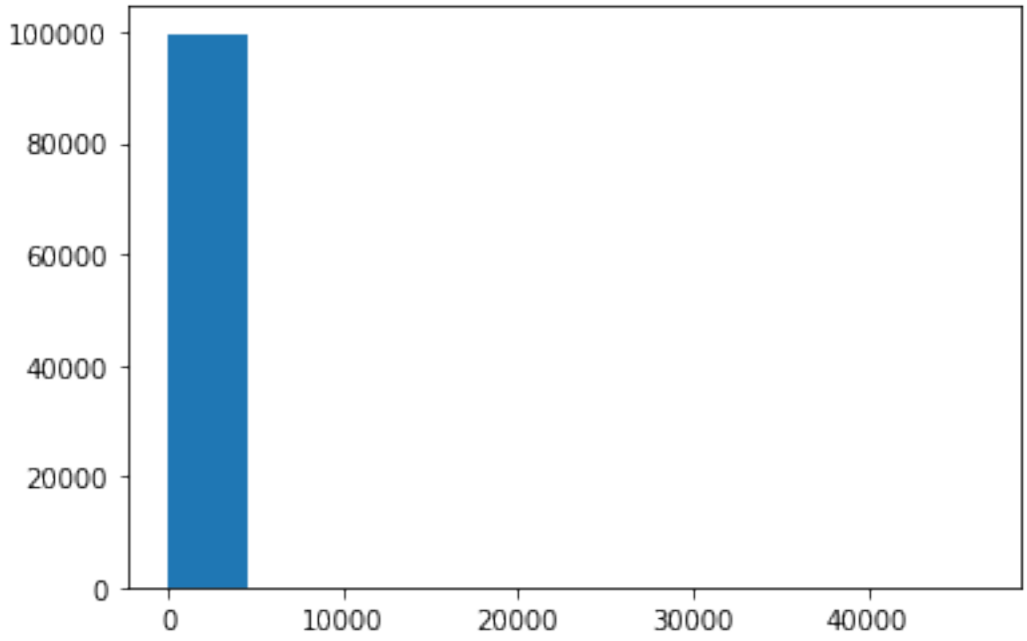
[ ]:
```
# set run = True if first time running, False otherwise as it takes a while␣
↪(upwards of an hour on some machines) to run
run = False

if run:
    pct_vals = []
    df = build_data()

    for i in tqdm(np.arange(100, 5001, 50)):
        X = fit_n_vocab(df=df, n_vocab=i)[0]
        pct = pct_no_words(X)
        pct_vals.append([i, pct])
    pct_df = pd.DataFrame(pct_vals, columns=['n_vocab', 'pct_words'])
    pct_df.to_csv('pct_df.csv')
else:
    pct_df = pd.read_csv('pct_df.csv')

plt.plot(pct_df['n_vocab'], pct_df['pct_words'])
plt.xlabel('Number of words in the bag of words')
plt.ylabel('Percent of obs. with 0 words from the bag of words')

plt.show()
```



We can see that there is a diminishing return effect once `N_VOCAB` reaches about the 1000 mark, so we will use a value within that range. (We ended up using 500 due to the amount of compute required for higher values) We realize that this is not a perfect solution as about 2% of the observations will have no information about them. However, with our limited compute we believe this is the best compromise.

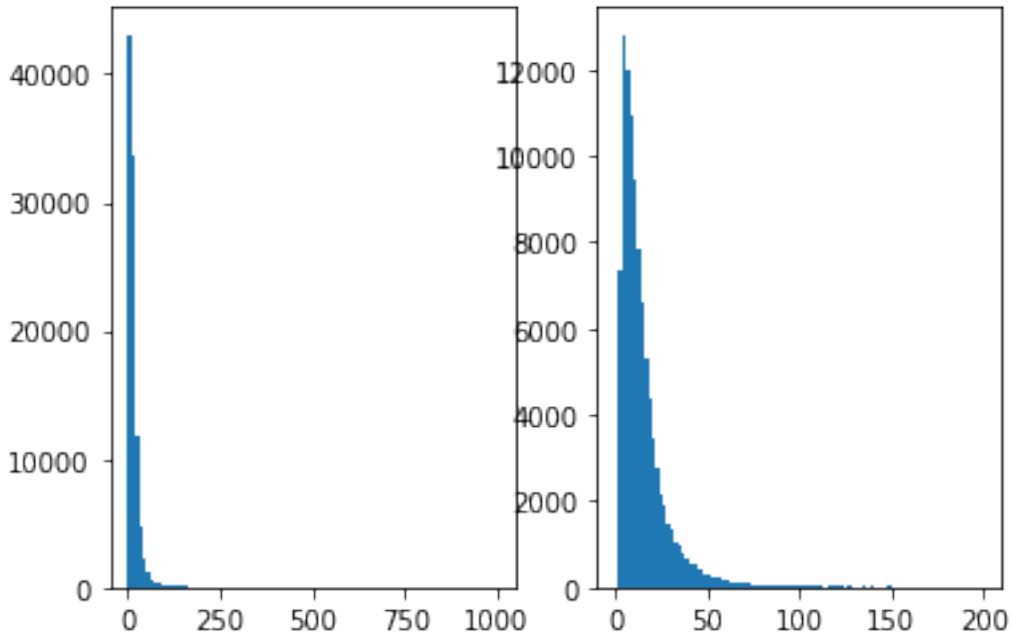
Next we will plot a histogram of our target variable, to see if any transformations should be made. We expect that the data will be positively skewed, as most of the posts on r/Askreddit garner little attention, and a few posts get most of the comments.

[ ]:
```
plt.hist(Y)
plt.show()
```



This plot basically meets our expectation of an extreme positive skewness. Let's bound the x-values and increase the number of bins.

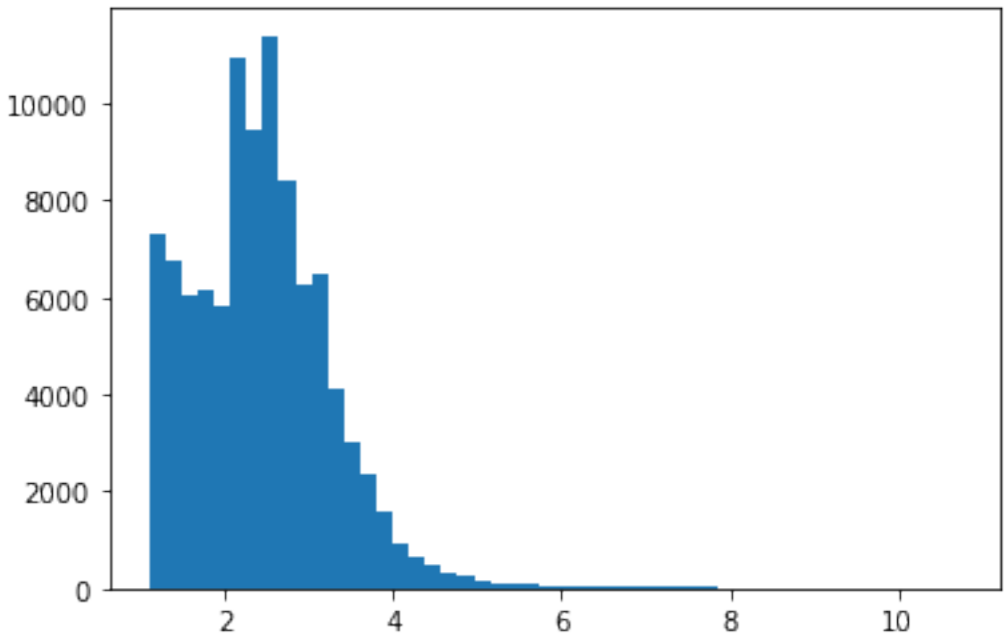
[ ]:
```
fig, ax = plt.subplots(1, 2)
ax[0].hist(Y, bins=100, range=(0, 1000))
ax[1].hist(Y, bins=100, range=(0, 200))
plt.show()
```



The skewness exists even when we set the upper bound to 200. A good transformation for skewed data is a log transform, so we will do that.

[ ]:
```
# Y transformation
log_Y = np.log(Y, dtype=float)
```

[ ]:
```
plt.hist(log_Y, bins=50)
plt.show()
```



We can see that the data is now much more normally distributed, although there still exists a positive skew. For the most part, this transformed Y is normal enough for the assumption of normality in models like linear regression.

1.8 Train, Validation, Test Split

[ ]:
```
df = pd.DataFrame(data = X)
df['Y'] = Y

# splitting the data into train and the remaining data
X_train = df.sample(frac = 0.8, random_state = 25)  # random_state serves as a␣
↪seed
X_remaining = df.drop(X_train.index)
```

```
Y_train = X_train.pop('Y')

#splitting the remaining data into validation and test
X_val = X_remaining.sample(frac = 0.5, random_state = 25)
X_test = X_remaining.drop(X_val.index)

Y_val = X_val.pop('Y')
Y_test = X_test.pop('Y')

scaler = StandardScaler()
scaler.fit(X_train)

print(f"No. of training examples: {X_train.shape[0]}")
print(f"No. of validation examples: {X_val.shape[0]}")
print(f"No. of testing examples: {X_test.shape[0]}")

del df, X_remaining
```

```
No. of training examples: 79822
No. of validation examples: 9978
No. of testing examples: 9977
```

1.9 Model Selection

One main focus for us is dimensionality reduction, since we have a large number of words as predictors. We will use PCA to attempt to reduce the dimensionality without losing important information.

We define `pca()` and `plot_pca()` to fit models with different numbers of principle components and plots the MSE.

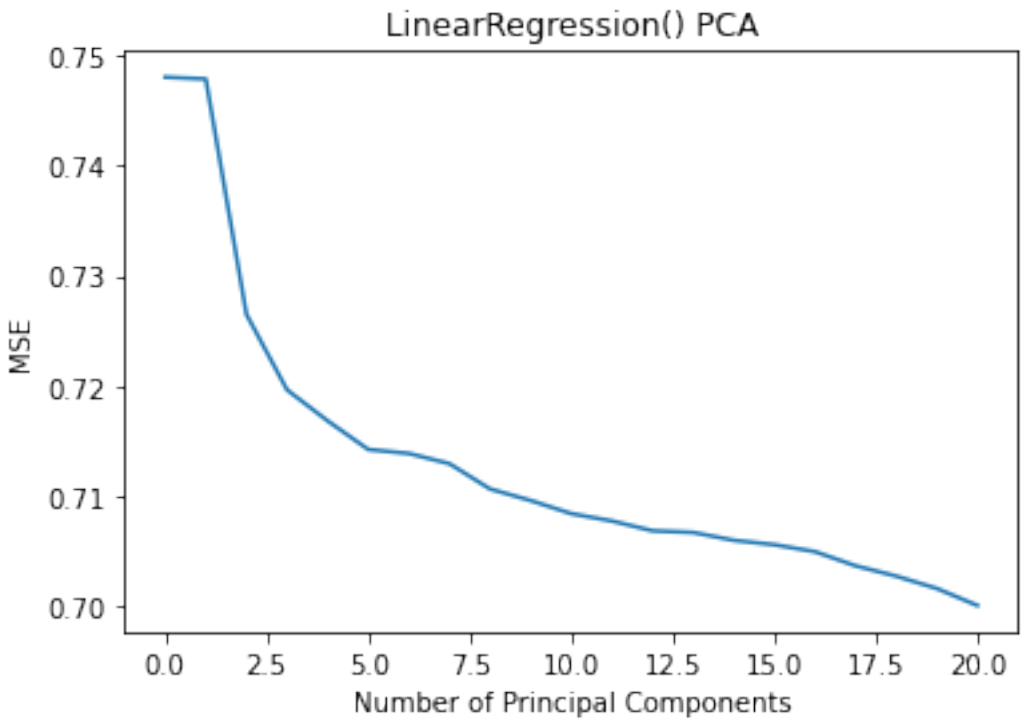
[ ]:
```
# code from: https://www.statology.org/
 ↪principal-components-regression-in-python/
def pca(model, X_train, Y_train):
    # Scale predictor variables to all have mean of 0 and standard deviation of␣
 ↪1. This ensures that no predictor variable is overly influencial in the␣
 ↪model if it happens to be measured in different units
    pca = PCA()
    X_reduced = pca.fit_transform(scaler.transform(X_train))

    # Define cross validation method. Here we are using k-fold cross-validation␣
 ↪with 10 folds repreated 3 times
    cv = RepeatedKFold(n_splits=10, n_repeats=3, random_state=1)

    mse = []
```

```
    # Calculate MSE with only the intercept
    score = -1 * cross_val_score(model,
                np.ones((len(X_reduced),1)), Y_train, cv=cv,
                scoring='neg_mean_squared_error').mean()
    mse.append(score)

    # Calculate MSE using cross-validation, adding one component at a time
    for i in tqdm(np.arange(1, N_VOCAB+1, int(N_VOCAB/20))):
        score = -1 * cross_val_score(model,
                X_reduced[:,:i], Y_train, cv=cv,␣
 ↪scoring='neg_mean_squared_error').mean()
        mse.append(score)
    return mse, pca

def plot_pca(model, X_train, Y_train):
    # Plot cross-validation results
    mse = pca(model, X_train, Y_train)
    plt.plot(mse)
    plt.xlabel('Number of Principal Components')
    plt.ylabel('MSE')
    plt.title((str(model) + ' PCA'))
    plt.show()
```

1.9.1 Linear Regression

[ ]:
```
plot_pca(LinearRegression(), X_train, np.log(Y_train))
```

```
100%|      | 20/20 [11:35<00:00, 34.79s/it]
```

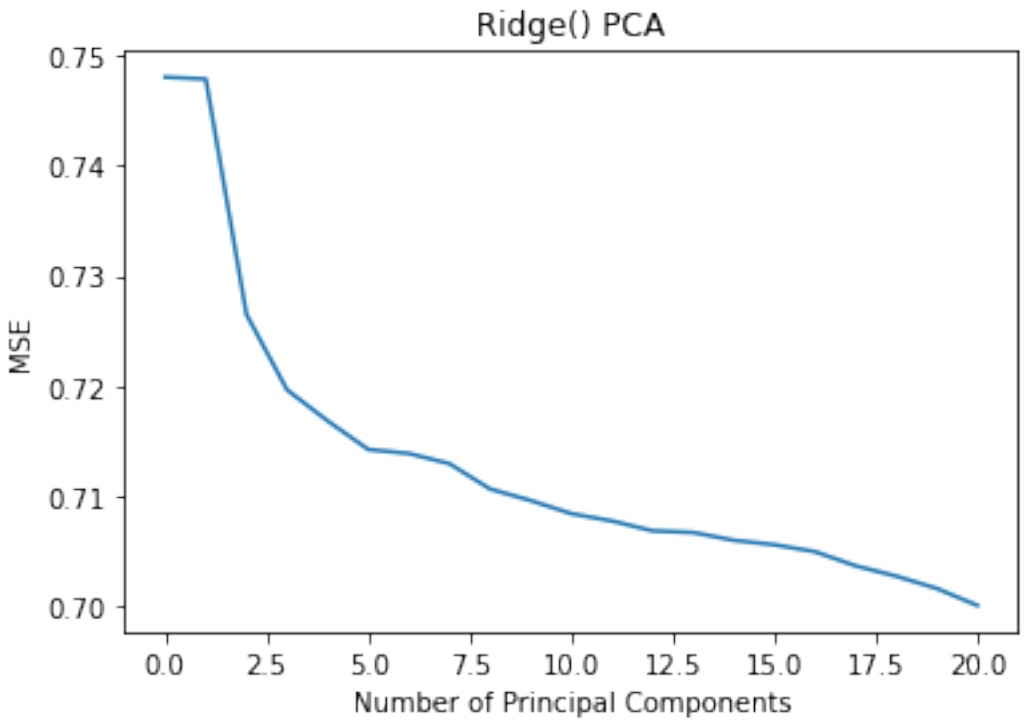


From the plot above, we can see that for linear regression the test MSE decreases with each component being added, and never increases. Therefore, we are going to use a model that uses all of the principal components.

1.9.2 Ridge Regression

[ ]:
```
plot_pca(Ridge(), X_train, np.log(Y_train))
```

```
100%|      | 20/20 [03:04<00:00,  9.21s/it]
```



From the plot above, we see that for ridge regression the MSE plot is extremely similar to that of linear regression, and here too the MSE only decreases, which means we will use all the principal components, or that every predictor will be used. As such, we will fit linear, ridge, and lasso regression.

[ ]:
```
scoreLR = cross_val_score(LinearRegression(), X_train, np.log(Y_train), cv = 5,
                          scoring = 'neg_mean_squared_error')
print(f'Cross Val MSE: {-1*scoreLR.mean()}')

lr = LinearRegression()
lr.fit(X_train, np.log(Y_train))
print(f'Validation MSE: {mean_squared_error(np.log(Y_val), lr.predict(X_val))}')
```

```
Cross Val MSE: 0.6987416384711121
Validation MSE: 0.7571649868879563
```

[ ]:
```
scoreRidge = cross_val_score(Ridge(), X_train, np.log(Y_train), cv = 5,
                          scoring = 'neg_mean_squared_error')
print(f'Cross Val MSE: {-1*scoreRidge.mean()}')

ridge = Ridge()
ridge.fit(X_train, np.log(Y_train))
print(f'Validation MSE: {mean_squared_error(np.log(Y_val), ridge.
 ↪predict(X_val))}')
```

```
Cross Val MSE: 0.6986898776081856
Validation MSE: 0.7571402947257673
```

1.9.3 Lasso Regression

For Lasso regression, we use a different approach and try to fit different α values with a grid search cross validation.

[ ]:
```
# code from: https://machinelearningmastery.com/lasso-regression-with-python/

# Define model
model = Lasso()

# Define model evaluation method
cv = RepeatedKFold(n_splits=10, n_repeats=3, random_state=1)

# Define grid
grid = dict()
grid['alpha'] = np.arange(0.01, 1, 0.01)

# Define search
search = GridSearchCV(model, grid, scoring='neg_mean_squared_error', cv=cv,␣
↪n_jobs=1, verbose=1)

# Perform the search
results = search.fit(X_train, np.log(Y_train))

# Summarize
print('MSE: %.3f' % results.best_score_)
print('Config: %s' % results.best_params_)
```

```
Fitting 30 folds for each of 99 candidates, totalling 2970 fits
MSE: -0.748
Config: {'alpha': 0.01}
```

[ ]:
```
print(f'Lasso Validation MSE: {mean_squared_error(results.predict(X_val), np.
↪log(Y_val))}')
```

```
Lasso Validation MSE: 0.8161913589323858
```

1.9.4 Selected Model

We get the following results for our validation set MSE:

Model	Validation MSE
Lasso Regression	0.8161913
Linear Regression	0.7571649
Ridge Regression	0.7571402

We see that ridge regression was able to have the least validation MSE, although the difference was slight. Therefore we select the ridge regression to be our final model.

We will now see how the ridge regression model performs on the test (hold-out) set.

[ ]:
```
print(f'Ridge Test MSE: {mean_squared_error(ridge.predict(X_test), np.
↪log(Y_test))}')
```

```
Ridge Test MSE: 0.8315668642581804
```

1.10 Interpretability

One important feature of simple regression models like linear regression and ridge regression is that the models and their parameters are extremely easy to interpret. This is especially helpful for our models as we can see which words contribute more positively to the number of comments and which words contribute negatively.

We (or this github post) create a function that can extract the coefficients (similar to R).

[ ]:
```
# code from: https://stackoverflow.com/questions/27928275/
↪find-p-value-significance-in-scikit-learn-linearregression
def make_lr_dict(m, y, p):
    params = np.append(m.intercept_,m.coef_)
    predictions = p

    newX = pd.DataFrame({"Constant":np.ones(len(X))}).join(pd.DataFrame(X))
    MSE = (sum((y-predictions)**2))/(len(newX)-len(newX.columns))

    # Note if you don't want to use a DataFrame replace the two lines above with
    # newX = np.append(np.ones((len(X),1)), X, axis=1)
    # MSE = (sum((y-predictions)**2))/(len(newX)-len(newX[0]))

    var_b = MSE*(np.linalg.inv(np.dot(newX.T,newX)).diagonal())
    sd_b = np.sqrt(var_b)
    ts_b = params/ sd_b

    sd_b = np.round(sd_b,3)
    ts_b = np.round(ts_b,3)
    #p_values = np.round(p_values,3)
    params = np.round(params,4)

    myDF3 = pd.DataFrame()
    myDF3["Coefficients"],myDF3["Standard Errors"],myDF3["t values"] =␣
↪[params,sd_b,ts_b]
    return myDF3.drop([0])
```

1.10.1 Linear Regression Coefficients

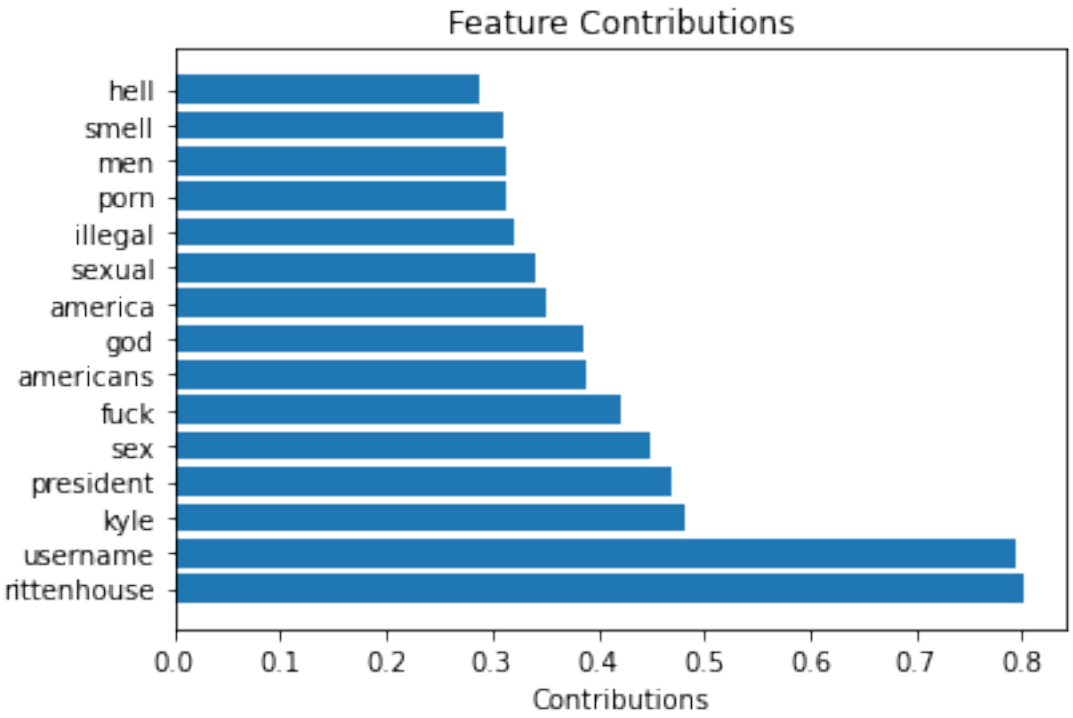
[ ]:
```
lr_pred = lr.predict(X_train)
lr_df = make_lr_dict(lr, np.log(Y_train), lr_pred)
lr_df['word'] = vocab
lr_df_sorted = lr_df.sort_values(by=['Coefficients'], ascending=False)
lr_df_sorted.head()
```

[ ]:

	Coefficients	Standard Errors	t values	word
364	0.8024	0.064	12.617	rittenhouse
458	0.7946	0.047	16.771	username
232	0.4826	0.071	6.784	kyle
338	0.4689	0.051	9.190	president
383	0.4488	0.009	47.585	sex

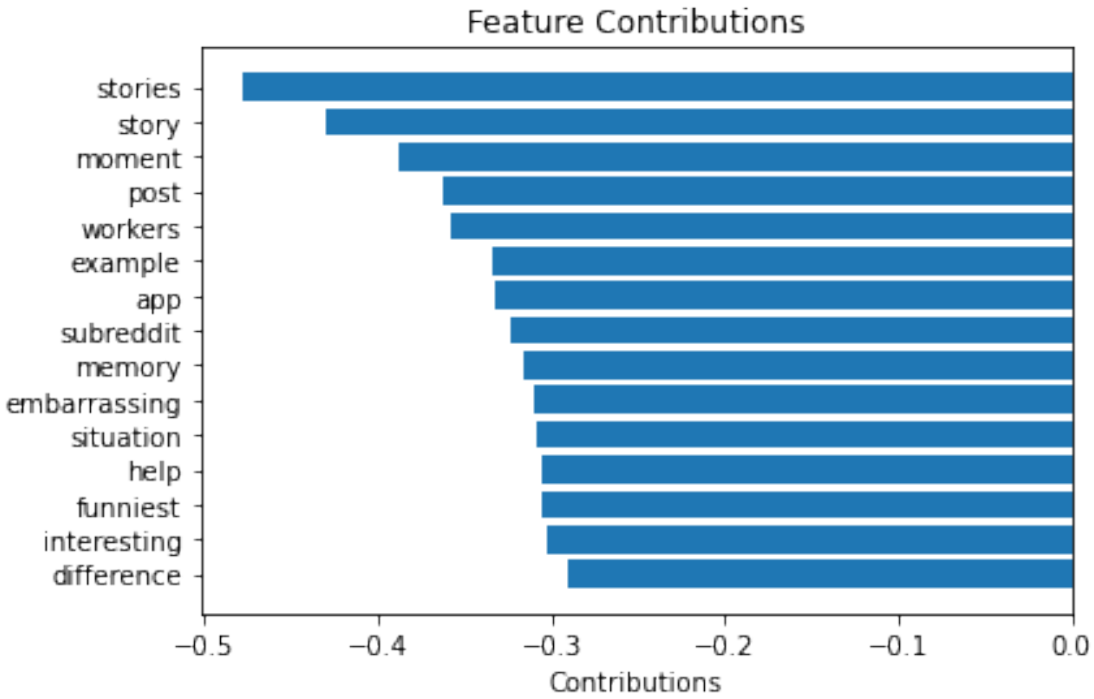
Positive Contributions

[ ]:
```
plt.barh([x for x in range(15)], lr_df_sorted['Coefficients'][0:15])
plt.yticks([x for x in range(15)], lr_df_sorted['word'][0:15])
plt.xlabel('Contributions')
plt.title('Feature Contributions')
plt.show()
```



Negative Contributions

[ ]:
```
plt.barh([x for x in range(15)], lr_df_sorted.tail(15)['Coefficients'])
plt.yticks([x for x in range(15)], lr_df_sorted.tail(15)['word'])
plt.xlabel('Contributions')
plt.title('Feature Contributions')
plt.show()
```



1.10.2 Ridge Regression Coefficients

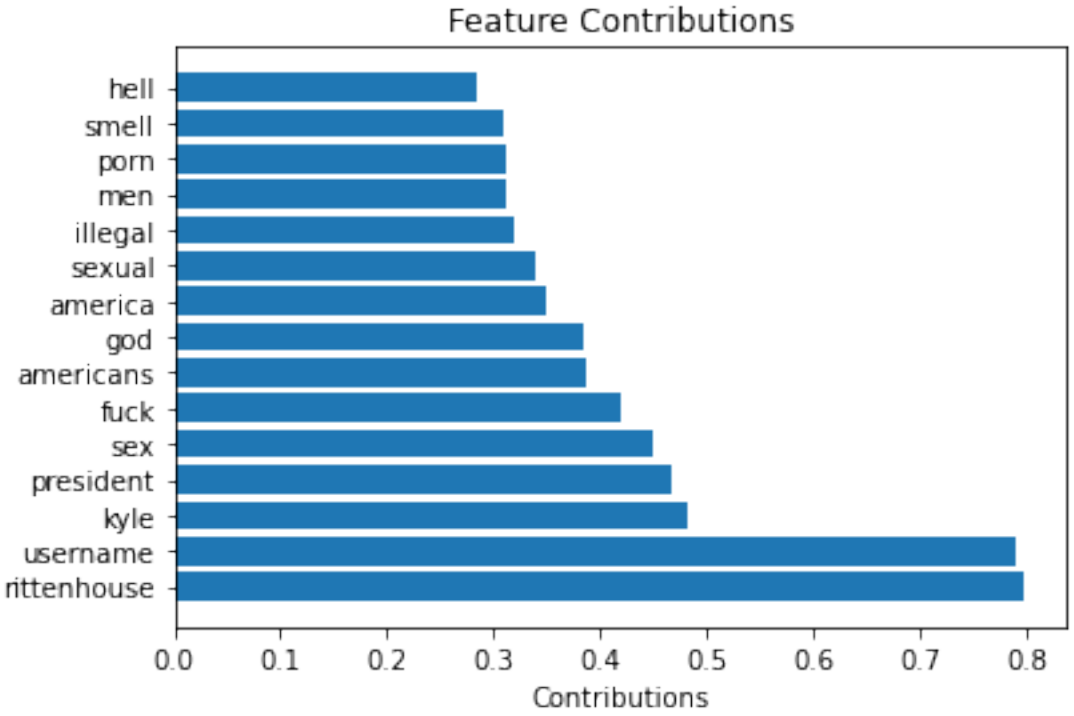
[ ]:
```
ridge_pred = ridge.predict(X_train)
ridge_df = make_lr_dict(ridge, np.log(Y_train), ridge_pred)
ridge_df['word'] = vocab
ridge_df_sorted = ridge_df.sort_values(by=['Coefficients'], ascending=False)
ridge_df_sorted.head()
```

[ ]:

	Coefficients	Standard Errors	t values	word
364	0.7987	0.064	12.560	rittenhouse
458	0.7904	0.047	16.683	username
232	0.4832	0.071	6.793	kyle
338	0.4664	0.051	9.142	president
383	0.4483	0.009	47.533	sex

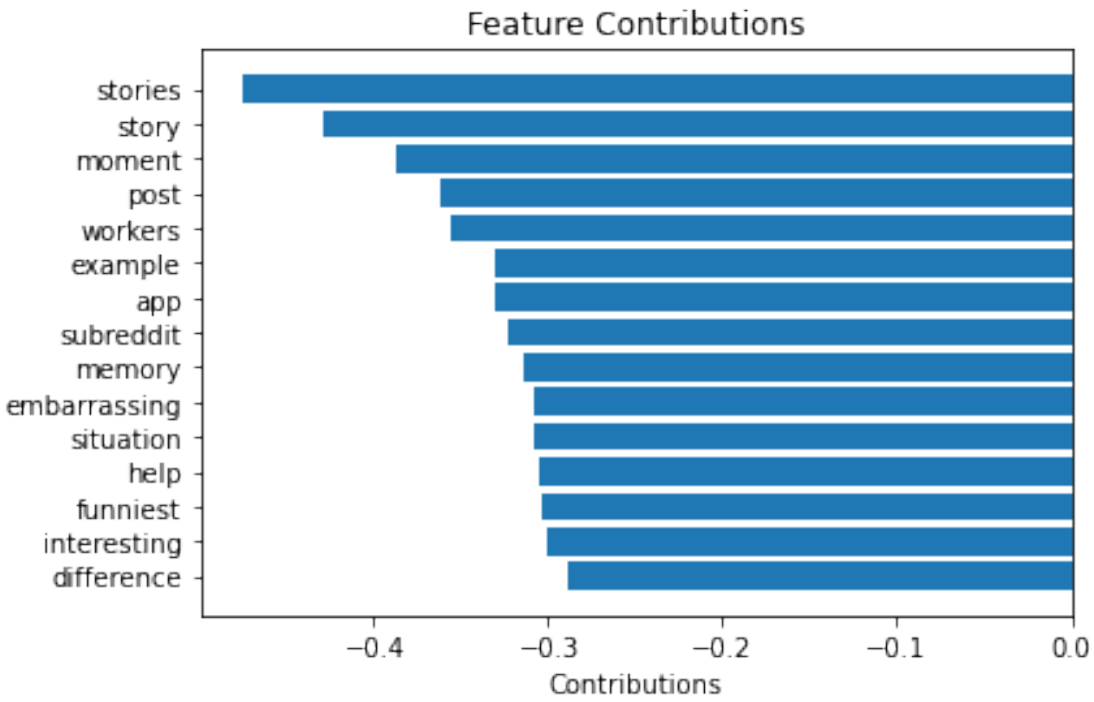
Positive Contributions

[ ]:
```
plt.barh([x for x in range(15)], ridge_df_sorted['Coefficients'][0:15])
plt.yticks([x for x in range(15)], ridge_df_sorted['word'][0:15])
plt.xlabel('Contributions')
plt.title('Feature Contributions')
plt.show()
```



Negative Contributions

[ ]:
```
plt.barh([x for x in range(15)], ridge_df_sorted.tail(15)['Coefficients'])
plt.yticks([x for x in range(15)], ridge_df_sorted.tail(15)['word'])
plt.xlabel('Contributions')
plt.title('Feature Contributions')
plt.show()
```



This would have been the most interesting part of the project if you're a regular Reddit user or if you frequent r/Askreddit! There are a few things of note here:

Since the model was trained on data between December 2021 and March 2022 (time of the Kyle Rittenhouse controversey), we can see that any mentions of "Kyle Rittenhouse" would have made your r/Askreddit post more popular than average. We can also see that any post mentioning anything of a sexual nature (big surprise, from the keywords "sex", "fuck", "sexual", "porn") would also have made your post most popular than average. We can also see the skew towards an American demographic through the words "president", "americans", and "america."

When comparing the linear regression model to the ridge model, we see that their predictions for the contirbutions of the different words are almost identical. The only difference we see when looking at the top 15 words, is that linear regression predicts that "porn" is more important than "men" and ridge predicts the opposite.

In addition, all the coefficients for the ridge regression are slightly smaller than those of the linear regression. This is expected because ridge regression penalizes for larger coefficients.

1.11 Number of Comments Predictions

We chose to predict the number of comments for different questions from the subreddit r/AskReddit using the ridge regression model. As all three models were extremely similar, the difference in prediction is minor between the different models.

[ ]:
```
np.exp(pred_from_str("What is the weirdest thing you've done while horny?",␣
↪ridge, vectorizer))
```

[ ]:
```
array([9.17218663])
```

This post received 16 comments, and our model predicted about 9.

[ ]:
```
np.exp(pred_from_str("Dear Americans of Reddit, how do you find this first year␣
↪of Biden's presidency compared to Trump's?", ridge, vectorizer))
```

[ ]:
```
array([12.72759948])
```

This questions received 24.5k comments, while our predictor only predicts for it to have about 13 comments.

When it comes to outliers, we can see that our model does not do a good job predicting the number of comments on a Reddit post in the subreddit r/AskReddit. This is probably due to the extremely skewed data and the ample random factors that exist which determine whether a post becomes popular or not. A way to improve our model could be to add predictors that depend on the time of the day and the day of the week the question was posted, and seeing if it improves the prediction.