playground

March 12, 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.metrics import mean_squared_error
from sklearn.model_selection import cross_val_score
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 = 100
```

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_7444\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)
```

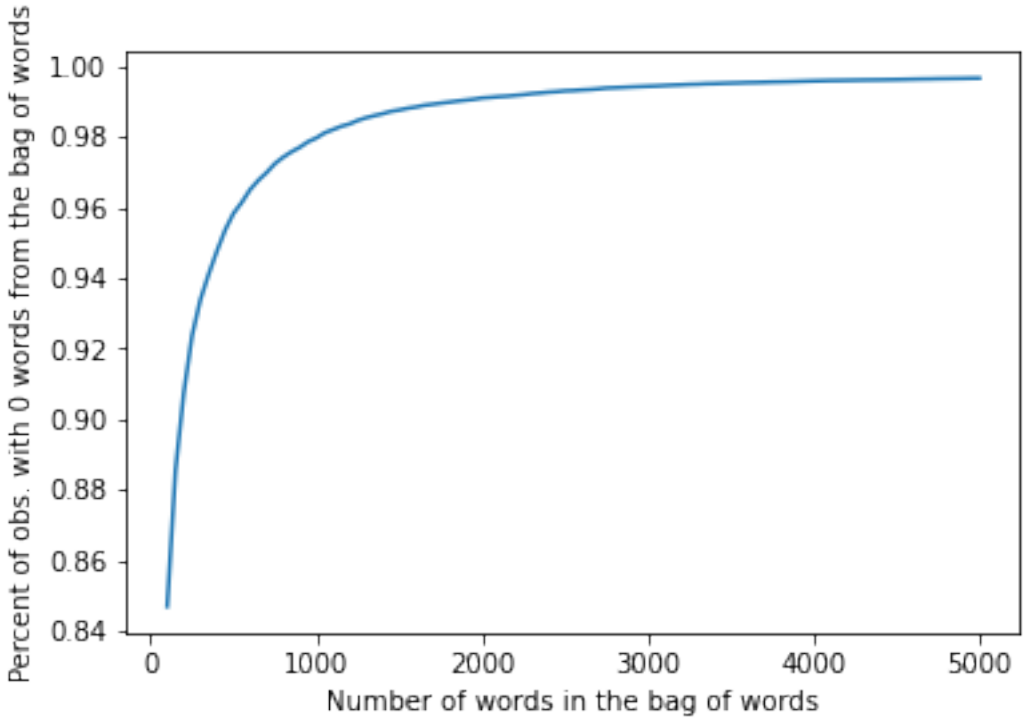
1.7 EDA

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.

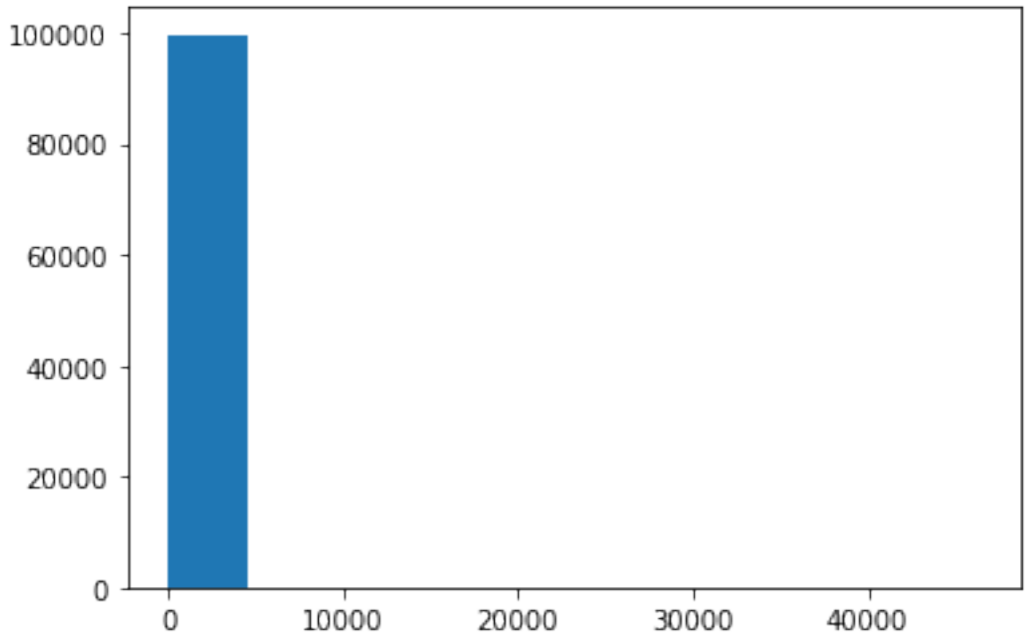
[ ]:
```
# set run = True if first time running, False otherwise as it takes a while␣
↪(upwards of an hour on some machines) to run
run = False

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 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.

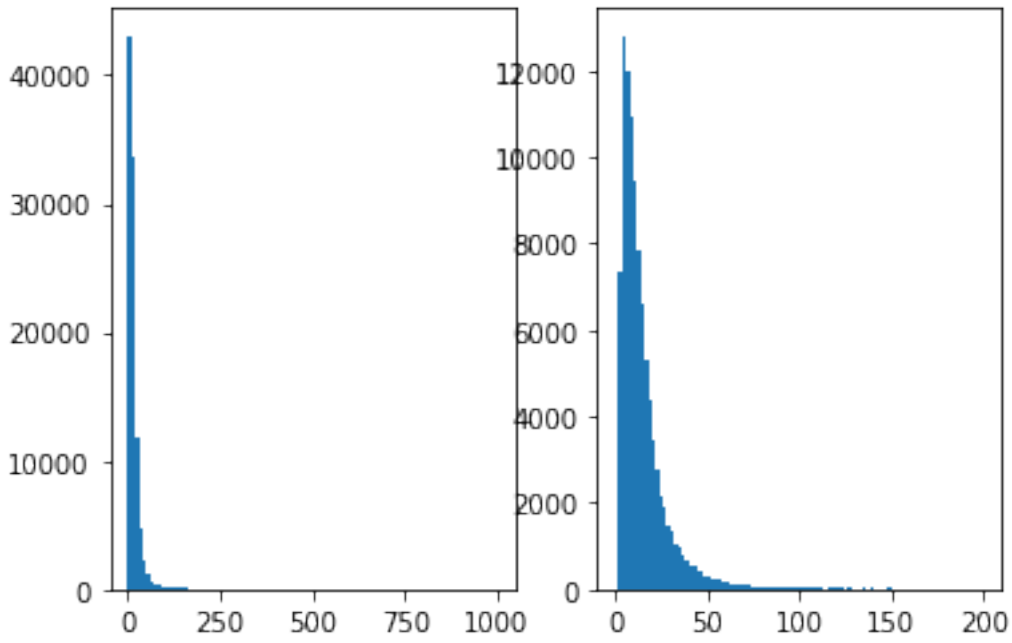
Next we will plot a histogram of our target variable, to see if any transformations should be made. We expect that the data will be positively skewed, as most of the posts on r/Askreddit garner little attention, and a few posts get most of the comments.

[ ]:
```
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.

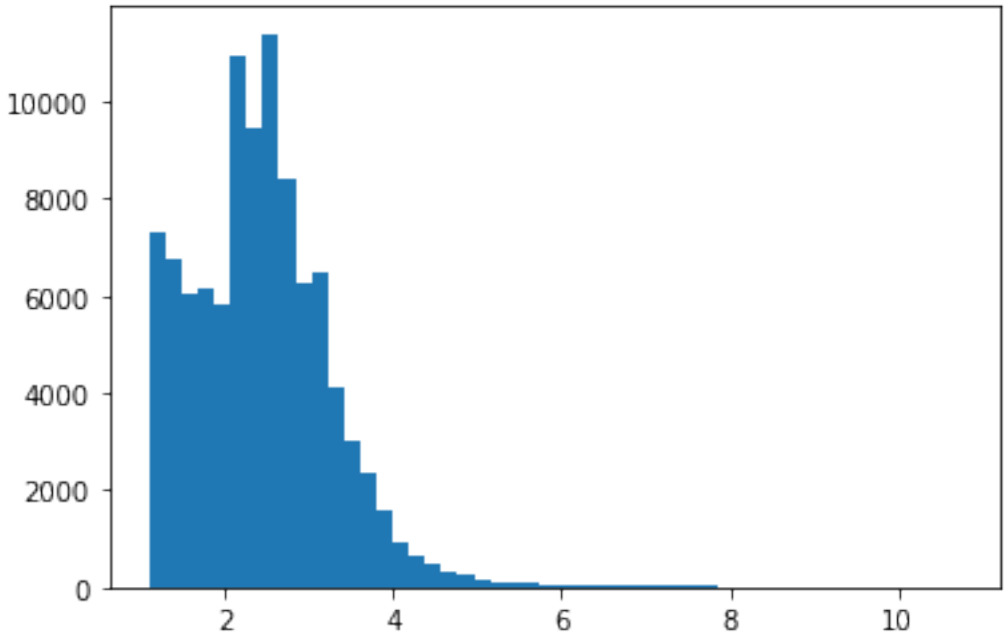
[ ]:
```
fig, ax = plt.subplots(1, 2)
ax[0].hist(Y, bins=100, range=(0, 1000))
ax[1].hist(Y, bins=100, range=(0, 200))
plt.show()
```



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(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.

[ ]:
```
import statsmodels.api as sm

ols = sm.OLS(Y, X)
ols_fit = ols.fit()

# https://stackoverflow.com/questions/51734180/converting-statsmodels-summary-object-to-pandas-dataframe
vocab_coeff_df = pd.DataFrame({'word': vocab, 'coeff': ols_fit.params, 'pval': ols_fit.pvalues})
```

[ ]:
```
vocab_coeff_df.sort_values('coeff', ascending=False)
```

[ ]:

	word	coeff	pval
54	opinion	1.835214	4.451592e-22
19	favourite	1.719135	5.712537e-21
81	thoughts	1.693542	5.618813e-21
5	biggest	1.687383	4.545969e-24
18	favorite	1.574314	5.476186e-59
..	…	…	…
75	take	0.247871	1.940359e-01
51	new	0.216357	2.445209e-01
24	friend	0.193925	1.816036e-01

```
10      could  0.179389  7.600573e-02
1        back  0.057469  6.884641e-01

[100 rows x 3 columns]
```

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

training_data = df.sample(frac = 0.8, random_state = 25)  # random_state serves␣
 ↪as a seed
testing_data = df.drop(training_data.index)

print(f"No. of training examples: {training_data.shape[0]}")
print(f"No. of testing examples: {testing_data.shape[0]}")

del df
```

```
No. of training examples: 7982
No. of testing examples: 1996
```

[ ]:
```
X_train = training_data.drop('Y', axis = 1)
X_test = testing_data.drop('Y', axis = 1)

Y_train = training_data['Y']
Y_test = testing_data['Y']

del training_data
del testing_data
```

[ ]:
```
from scipy import stats

# 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(linReg.intercept_,linReg.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
```

```
    p_values =[2*(1-stats.t.cdf(np.abs(i),(len(newX)-len(newX[0])))) for i in␣
↪ts_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"],myDF3["Probabilities"] = [params,sd_b,ts_b,p_values]
    return myDF3.drop([0])
```

[ ]:
```
from sklearn.ensemble import RandomForestRegressor

randForest = RandomForestRegressor().fit(X_train, Y_train)
scores = cross_val_score(randForest, X_train, Y_train, cv = 5)
scores
```

[ ]:
```
array([-0.13508352, -0.12277632, -0.06752203, -0.10442074, -0.11076994])
```

[ ]:
```
from sklearn.linear_model import LinearRegression

linReg = LinearRegression().fit(X_train, Y_train)
lin_ypred = linReg.predict(X_train)
print(mean_squared_error(Y_train, lin_ypred))

scorelin = cross_val_score(LinearRegression(), X_train, Y_train, cv = 5,
                           scoring = 'neg_mean_squared_error')
print(-1*scorelin.mean())
```

```
0.676335929737053
0.6941848249960213
```

[ ]:
```
lr_df = make_lr_dict(linReg, Y_train, lin_ypred)
lr_df['word'] = vocab
lr_df.sort_values(by=['Coefficients'], ascending=False)
```

[ ]:

	Coefficients	Standard Errors	t values	Probabilities	word
69	0.6239	0.069	9.028	NaN	sex
55	0.2262	0.083	2.729	NaN	opinion
20	0.2147	0.080	2.690	NaN	favourite
53	0.2096	0.079	2.670	NaN	old
86	0.1828	0.082	2.231	NaN	us
..	…	…	…	…	…
52	-0.2541	0.081	-3.141	NaN	new
18	-0.2583	0.069	-3.751	NaN	family

```
17       -0.2974            0.075    -3.980            NaN  experience
46       -0.4353            0.067    -6.468            NaN      moment
75       -0.4870            0.071    -6.873            NaN       story

[100 rows x 5 columns]
```

[ ]:
```
lr_df
```

[ ]:
```
     Coefficients  Standard Errors  t values  Probabilities
0          2.4888            0.012   199.614            NaN
1         -0.0990            0.062    -1.601            NaN
2          0.0228            0.062     0.366            NaN
3         -0.0657            0.080    -0.819            NaN
4         -0.0915            0.036    -2.517            NaN
..            …                …         …              …
96        -0.0514            0.049    -1.059            NaN
97         0.0144            0.046     0.311            NaN
98         0.0074            0.023     0.324            NaN
99        -0.1895            0.063    -3.012            NaN
100        0.0609            0.070     0.874            NaN

[101 rows x 4 columns]
```

[ ]:
```
df
```

[ ]:
```
from sklearn.linear_model import Lasso

lassoReg = Lasso().fit(X_train, Y_train)
lassoReg.score(X_train, Y_train)
```

[ ]:
```
0.0
```

[ ]:
```
from sklearn.linear_model import Ridge

ridgeReg = Ridge().fit(X_train, Y_train)
ridge_ypred = ridgeReg.predict(X_train)
print(mean_squared_error(Y_train, ridge_ypred))

scoreridge = cross_val_score(Ridge(), X_train, Y_train, cv = 5,
                             scoring = 'neg_mean_squared_error')
print(-1*scoreridge.mean())
```

```
0.6763387097664976
0.6939658846812932
```

[ ]:
```
import statsmodels.api as sm

ols = sm.OLS(Y_train, X_train)
```

```
ols_fit = ols.fit()

ols_ypred = ols_fit.predict(X_train)
print(mean_squared_error(Y_train, ols_ypred))
```

```
2.828248869367714
```

[ ]:
```
del ols
del ols_fit
```

[ ]:
```
from sklearn.neural_network import MLPRegressor

mlpReg = MLPRegressor(max_iter = 500, random_state = 1).fit(X_train, Y_train)
#print(mlpReg.score(X_train, Y_train))

ypred = mlpReg.predict(X_train)
print(mean_squared_error(Y_train, ypred))

scoreMLP = cross_val_score(MLPRegressor(max_iter = 500, random_state = 1),␣
 ↪X_train, Y_train, cv = 5,
                           scoring = 'neg_mean_squared_error')
print(-1*scoreMLP.mean())
```

```
0.4232551898203105
0.9373989289273446
```

[ ]:
```
from sklearn.neighbors import KNeighborsRegressor

kReg = KNeighborsRegressor().fit(X_train, Y_train)
k_ypred = kReg.predict(X_train)
print(mean_squared_error(Y_train, k_ypred))

scoreK = cross_val_score(KNeighborsRegressor(), X_train, Y_train, cv = 5,
                           scoring = 'neg_mean_squared_error')
print(-1*scoreK.mean())
```

```
0.7199426985446578
0.8282334772548527
```

[ ]:
```
X_train.shape, Y_train.shape
```

[ ]:
```
((159644, 500), (159644,))
```

[ ]:
```
X_test.shape, Y_test.shape
```

[ ]:
```
((39911, 500), (39911,))
```

[ ]:
```

```