In [1]:
```
#
#Packages and Imports
#
```

In [2]:
```
import os
import pandas
import re
import math
from nltk.corpus import stopwords
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.linear_model import LogisticRegression
import numpy
import matplotlib.pyplot as plt
```

In [3]:
```
#
#Reusable functions
#
```

In [4]:
```
def read_input(input_path:str) -> str:
    file_data = open(input_path , 'r')
    return file_data.read()
```

In [5]:
```
def line_break_tokenizer(input:str):
    return input.split('\n\n')

def word_count_tokenizer(text_col, count):
    result = []
    for i in range(0, len(text_col), count):
        result = result + [text_col[i:i + count]]

    return result

def clean_text(text:str):
    clean = re.sub('[\W_]+', ' ', text.lower())
    clean = re.sub('[\d]+', ' ', clean)
    return re.sub(' +', ' ', clean)
```

In [6]:
```
#
#TF-IDF Calculation
#
```

In [7]:
```
def computeTF(word_list, doc_size):
    tfDict = [{}]
    for i  in range(0, len(word_list)):
        dicts = {}
        for word, count in word_list[i].items():
            dicts[word] = count / float(doc_size)
        tfDict.append(dicts)
    return tfDict

def computeIDF(documents, final_word_list):
    N = len(documents)

    idfDict = dict.fromkeys(final_word_list, 0)
    for document in documents:
        for word, val in document.items():
            if val > 0:
                idfDict[word] += 1

    for word, val in idfDict.items():
        if(val != 0):
            idfDict[word] = float(math.log(float(N) / float(val)))
        else:
            idfDict[word] = 0
    return idfDict

def computeTFIDF(doc_word, idfs, key):
    tfidf = [{}]
    for i  in range(0, len(doc_word)):
        dicts = {}
        for word, val in doc_word[i].items():
            dicts[word] = val * idfs[word]
        dicts['123'] = key
        tfidf.append(dicts)
    return tfidf
```

In [8]:
```
#
#Logistic Regression
#
```

In [38]:
```
def sigmoid(val):
    return 1 / (1 + numpy.exp(-val))

def weight(theta, x):
    return numpy.dot(x, theta)

def probability(theta, x):
    return sigmoid(weight(theta,x))

def cost(theta, x, y):
    matrix = x.shape[0]
    net_cost = -(1 / matrix) * numpy.sum( y * numpy.log(probability(thet
a, x)) + (1 - y) * numpy.log( 1 - probability(theta, x)))
    return net_cost

def gradient(theta, x, y):
    matrix = x.shape[0]
    return (1 / matrix) * (-numpy.dot(x.T, probability(theta ,x) - y))

def fit(x, y, max_step=500, alpha=.05):
    x = numpy.insert(x, 0, 1, axis=1)
    thetas = []
    classification = numpy.unique(y)
    costs = numpy.zeros(max_step)

    for c in classification:
        binary_y = numpy.where(y == c, 1, 0)
        theta = numpy.zeros(x.shape[1])
        for epoch in range(max_step):
            costs[epoch] = cost(theta, x, binary_y)
            theta = theta + alpha * gradient(theta, x, y)

        thetas.append(theta)

    return thetas, classification, costs

def predict(classification, thetas, x):
    x = numpy.insert(x, 0, 1, axis=1)
    prediction = [numpy.argmax([probability(xi, theta) for theta in thet
as]) for xi in x]
    return [classification[p] for p in prediction]
```

In [10]:
```
set_word = set(stopwords.words('english'))
```

In [11]:
```
#
#Fyodor Dostoyevsky
#
```

In [12]:
```
data_folder = './data/'
file_nameA = 'FDBodyOnly.txt'
input_pathA = os.path.join(data_folder, file_nameA)
```

In [13]:
```
textA = read_input(input_pathA)
all_wordA = clean_text(textA).split()
tokenA = word_count_tokenizer(all_wordA, 2801)
allA = set(all_wordA)

train_tokenA = []
valid_tokenA = []
train_sizeA = 0
valid_sizeA = 0

#Split into train/validation
for s in range(len(tokenA)):
    if(s % 8 != 0):
        train_tokenA += [tokenA[s]]
        train_sizeA += len(tokenA[s])
    else:
        valid_tokenA += [tokenA[s]]
        valid_sizeA += len(tokenA[s])

word_colA = []
vword_colA = []

#find unquie wordset for TFIDF
for para in train_tokenA:
    word_colA = set(word_colA).union(set(para))

for para in valid_tokenA:
    vword_colA = set(vword_colA).union(set(para))

train_sampleA = [{}]
valid_sampleA = [{}]

for para in train_tokenA:
    word_countA = dict.fromkeys(allA, 0)
    for word in para:
        #if(word not in set_word):
            word_countA[word] += 1
    train_sampleA += [word_countA]

for para in valid_tokenA:
    vword_countA = dict.fromkeys(allA, 0)
    for word in para:
        #if(word not in set_word):
            vword_countA[word] += 1
    valid_sampleA += [vword_countA]


# remove Setwords
# for sword in set_word:
#     try:
#         del word_countA[sword]
#         word_colA = list(filter(lambda a: a != sword, word_colA))
#         all_wordA = list(filter(lambda a: a != sword, all_wordA))
#     except Exception: pass
```

In [14]:
```
print(len(all_wordA)/128)
```

```
2802.0546875
```

In [15]:
```
tfA = computeTF(train_sampleA, train_sizeA)
vtfA = computeTF(valid_sampleA, valid_sizeA)
```

In [ ]:
```
```

In [16]:
```
#
#Arthur Conan Doyle
#
```

In [17]:
```
file_nameB = 'ACDBodyOnly.txt'
input_pathB = os.path.join(data_folder, file_nameB)
```

In [18]:
```
textB = read_input(input_pathB)
all_wordB = clean_text(textB).split()
tokenB = word_count_tokenizer(all_wordB, 825)
allB = set(all_wordB)

train_tokenB = []
valid_tokenB = []
train_sizeB = 0
valid_sizeB = 0

for s in range(len(tokenB)):
    if(s % 8 != 0):
        train_tokenB += [tokenB[s]]
        train_sizeB += len(tokenB[s])
    else:
        valid_tokenB += [tokenB[s]]
        valid_sizeB += len(tokenB[s])

word_colB = []
vword_colB = []

for para in train_tokenB:
    word_colB = set(word_colB).union(set(para))

for para in valid_tokenB:
    vword_colB = set(vword_colB).union(set(para))

train_sampleB = [{}]
valid_sampleB = [{}]

for para in train_tokenB:
    word_countB = dict.fromkeys(allB, 0)
    for word in para:
        #if(word not in set_word):
            word_countB[word] += 1
    train_sampleB += [word_countB]

for para in valid_tokenB:
    vword_countB = dict.fromkeys(allB, 0)
    for word in para:
        #if(word not in set_word):
            vword_countB[word] += 1
    valid_sampleB += [vword_countB]
# for sword in set_word:
#     try:
#         del word_countB[sword]
#         word_colB = list(filter(lambda a: a != sword, word_colB))
#         all_wordB = list(filter(lambda a: a != sword, all_wordB))
#     except Exception: pass
```

In [19]:
```
print(len(all_wordB)/128)
```

```
825.953125
```

In [20]:
```
tfB = computeTF(train_sampleB, train_sizeB)
vtfB = computeTF(valid_sampleB, valid_sizeB)
```

In [ ]:

In [21]:
```
#
#Jane Austen
#
```

In [22]:
```
file_nameC = 'JABodyOnly.txt'
input_pathC = os.path.join(data_folder, file_nameC)
```

In [23]:
```
textC = read_input(input_pathC)
all_wordC = clean_text(textC).split()
tokenC = word_count_tokenizer(all_wordC, 6161)
allC = set(all_wordC)

train_tokenC = []
valid_tokenC = []
train_sizeC = 0
valid_sizeC = 0

for s in range(len(tokenC)):
    if(s % 8 != 0):
        train_tokenC += [tokenC[s]]
        train_sizeC += len(tokenC[s])
    else:
        valid_tokenC += [tokenC[s]]
        valid_sizeC += len(tokenC[s])

word_colC = []
vword_colC = []

for para in train_tokenC:
    word_colC = set(word_colC).union(set(para))

for para in valid_tokenC:
    vword_colC = set(vword_colC).union(set(para))

train_sampleC = [{}]
valid_sampleC = [{}]

for para in train_tokenC:
    word_countC = dict.fromkeys(allC, 0)
    for word in para:
        #if(word not in set_word):
            word_countC[word] += 1
    train_sampleC += [word_countC]

for para in valid_tokenC:
    vword_countC = dict.fromkeys(allC, 0)
    for word in para:
        #if(word not in set_word):
            vword_countC[word] += 1
    valid_sampleC += [vword_countC]
# for sword in set_word:
#     try:
#         del word_countC[sword]
#         word_colC= list(filter(lambda a: a != sword, word_colC))
#         all_wordC = list(filter(lambda a: a != sword, all_wordC))
#     except Exception: pass
```

In [24]:
```
print(len(all_wordC)/128)
```

```
6161.8125
```

In [25]:
```
tfC = computeTF(train_sampleC, train_sizeC)
vtfC = computeTF(valid_sampleC, valid_sizeC)
```

In [ ]:

In [26]:
```
#END
```

In [27]:
```
final_words = set(allA).union(set(allB)).union(set(allC))
idfs = computeIDF(train_sampleA + train_sampleB + train_sampleC, final_w
ords)

vfinal_words = set(vword_colA).union(set(vword_colB)).union(set(vword_co
lC))
vidfs = computeIDF(valid_sampleA + valid_sampleB + valid_sampleC, final_
words)
```

In [28]:
```
tfidfA = computeTFIDF(tfA, idfs, 0)
tfidfB = computeTFIDF(tfB, idfs, 1)
tfidfC = computeTFIDF(tfC, idfs, 2)
df = pandas.DataFrame(tfidfA + tfidfB + tfidfC)

vtfidfA = computeTFIDF(vtfA, vidfs, 0)
vtfidfB = computeTFIDF(vtfB, vidfs, 1)
vtfidfC = computeTFIDF(vtfC, vidfs, 2)
vdf = pandas.DataFrame(vtfidfA + vtfidfB + vtfidfC)
```

In [29]:
```
print(len(tfidfA))
print(len(vtfidfA))
```

```
115
20
```

In [30]:
```
vectorizer = TfidfVectorizer(stop_words='english')
vectors = vectorizer.fit_transform([clean_text(textA), clean_text(textB
), clean_text(textC)])
feature_names = vectorizer.get_feature_names()
dense = vectors.todense()
denselist = dense.tolist()
df2 = pandas.DataFrame(denselist, columns=feature_names)
```

In [31]:
```
print(df)
```

```
     123  pies   silence  song  incapable  slur  formally  imperative
sweat  \
0    NaN   NaN       NaN   NaN        NaN   NaN       NaN         NaN
NaN
1    0.0   NaN       NaN   NaN        NaN   NaN       NaN         NaN
NaN
2    0.0   NaN       NaN   NaN        NaN   NaN       NaN         NaN
NaN
3    0.0   0.0  0.000000   0.0        0.0   0.0  0.000000         0.0
0.0
4    0.0   0.0  0.000010   0.0        0.0   0.0  0.000000         0.0
0.0
..   ...   ...       ...   ...        ...   ...       ...         ...
...
340  2.0   0.0  0.000001   0.0        0.0   NaN  0.000000         NaN
0.0
341  2.0   0.0  0.000000   0.0        0.0   NaN  0.000013         NaN
0.0
342  2.0   0.0  0.000003   0.0        0.0   NaN  0.000000         NaN
0.0
343  2.0   0.0  0.000007   0.0        0.0   NaN  0.000000         NaN
0.0
344  2.0   0.0  0.000000   0.0        0.0   NaN  0.000000         NaN
0.0

     establishment  ...  ebony  incur  beet  switch  misspent  surprize
s  \
0              NaN  ...    NaN    NaN   NaN     NaN       NaN        Na
N
1              NaN  ...    NaN    NaN   NaN     NaN       NaN        Na
N
2              NaN  ...    NaN    NaN   NaN     NaN       NaN        Na
N
3         0.000000  ...    NaN    NaN   NaN     NaN       NaN        Na
N
4         0.000000  ...    NaN    NaN   NaN     NaN       NaN        Na
N
..             ...  ...    ...    ...   ...     ...       ...
...
340       0.000000  ...    0.0    0.0   0.0     0.0       0.0        0.
0
341       0.000000  ...    0.0    0.0   0.0     0.0       0.0        0.
0
342       0.000000  ...    0.0    0.0   0.0     0.0       0.0        0.
0
343       0.000004  ...    0.0    0.0   0.0     0.0       0.0        0.
0
344       0.000000  ...    0.0    0.0   0.0     0.0       0.0        0.
0

     fraser  alcove  invalids  airing
0       NaN     NaN       NaN     NaN
1       NaN     NaN       NaN     NaN
2       NaN     NaN       NaN     NaN
3       NaN     NaN       NaN     NaN
4       NaN     NaN       NaN     NaN
..      ...     ...       ...     ...
```

```
340      0.0      0.0       0.0      0.0
341      0.0      0.0       0.0      0.0
342      0.0      0.0       0.0      0.0
343      0.0      0.0       0.0      0.0
344      0.0      0.0       0.0      0.0

[345 rows x 20939 columns]
```

In [32]:
```
df.fillna(0, inplace=True)
vdf.fillna(0, inplace=True)
```

In [33]:
```
logreg = LogisticRegression()
logreg.fit(df[final_words], df['123'])
```

Out[33]:
```
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=
True,
                   intercept_scaling=1, l1_ratio=None, max_iter=100,
                   multi_class='auto', n_jobs=None, penalty='l2',
                   random_state=None, solver='lbfgs', tol=0.0001, verbo
se=0,
                   warm_start=False)
```

In [34]:
```
ans = logreg.predict(vdf[final_words])
print(ans)
```

```
[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.
 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
```

In [35]:
```
print(vdf['123'].values)
```

```
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 1.
1.
 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 0. 2. 2. 2. 2. 2. 2.
2.
 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2.]
```

In [39]:
```
df_x = df[final_words].values
df_y = df['123']
th, cl, cost = fit(df_x, df_y)
print("Theta: ")
print(th)
print("Classification: ")
print(cl)
print(cost)
```

```
Theta:
[array([ 3.02202031e+00,  4.01152328e-06,  2.01457002e-06, ...,
       -1.17136136e-06, -6.43964950e-06,  1.71072697e-07]), array([ 3.0
2202031e+00,  4.01152328e-06,  2.01457002e-06, ...,
       -1.17136136e-06, -6.43964950e-06,  1.71072697e-07]), array([ 3.0
2202031e+00,  4.01152328e-06,  2.01457002e-06, ...,
       -1.17136136e-06, -6.43964950e-06,  1.71072697e-07])]
Classification:
[0. 1. 2.]
[0.69314718 0.69738802 0.70172386 0.70614981 0.71066108 0.71525304
 0.71992117 0.72466111 0.72946864 0.73433965 0.73927021 0.74425647
 0.74929476 0.75438153 0.75951334 0.7646869  0.76989904 0.7751467
 0.78042697 0.78573701 0.79107415 0.79643578 0.80181943 0.80722273
 0.8126434  0.81807928 0.82352828 0.82898842 0.83445781 0.83993463
 0.84541717 0.85090378 0.85639289 0.86188301 0.86737273 0.8728607
 0.87834563 0.8838263  0.88930156 0.89477032 0.90023152 0.90568419
 0.91112738 0.91656023 0.92198188 0.92739156 0.93278852 0.93817205
 0.94354149 0.94889622 0.95423564 0.95955922 0.96486642 0.97015677
 0.9754298  0.9806851  0.98592226 0.99114091 0.99634072 1.00152135
 1.00668251 1.01182392 1.01694534 1.02204652 1.02712725 1.03218733
 1.03722659 1.04224486 1.047242   1.05221787 1.05717235 1.06210535
 1.06701676 1.07190652 1.07677455 1.08162079 1.08644521 1.09124777
 1.09602843 1.10078719 1.10552403 1.11023896 1.11493197 1.11960309
 1.12425233 1.12887973 1.13348531 1.13806912 1.1426312  1.1471716
 1.15169038 1.15618759 1.1606633  1.16511758 1.16955049 1.17396212
 1.17835255 1.18272185 1.1870701  1.19139741 1.19570385 1.19998953
 1.20425453 1.20849895 1.21272289 1.21692646 1.22110974 1.22527286
 1.22941591 1.23353901 1.23764225 1.24172574 1.24578961 1.24983395
 1.25385889 1.25786452 1.26185097 1.26581835 1.26976678 1.27369636
 1.27760721 1.28149945 1.28537319 1.28922855 1.29306564 1.29688458
 1.30068548 1.30446847 1.30823365 1.31198114 1.31571105 1.31942351
 1.32311861 1.32679649 1.33045725 1.33410101 1.33772788 1.34133797
 1.3449314  1.34850828 1.35206871 1.35561282 1.35914071 1.36265249
 1.36614826 1.36962815 1.37309226 1.3765407  1.37997357 1.38339098
 1.38679303 1.39017984 1.39355151 1.39690814 1.40024983 1.40357669
 1.40688882 1.41018632 1.41346929 1.41673784 1.41999206 1.42323205
 1.42645791 1.42966974 1.43286763 1.43605169 1.43922199 1.44237865
 1.44552175 1.44865139 1.45176766 1.45487066 1.45796046 1.46103717
 1.46410088 1.46715167 1.47018962 1.47321484 1.47622741 1.47922741
 1.48221492 1.48519004 1.48815285 1.49110344 1.49404187 1.49696825
 1.49988264 1.50278513 1.50567581 1.50855474 1.51142202 1.51427771
 1.5171219  1.51995466 1.52277607 1.52558621 1.52838515 1.53117296
 1.53394973 1.53671551 1.53947039 1.54221444 1.54494773 1.54767033
 1.55038232 1.55308375 1.5557747  1.55845524 1.56112544 1.56378537
 1.56643508 1.56907466 1.57170416 1.57432364 1.57693319 1.57953285
 1.58212269 1.58470277 1.58727317 1.58983393 1.59238513 1.59492682
 1.59745906 1.59998192 1.60249545 1.60499971 1.60749476 1.60998066
 1.61245746 1.61492523 1.61738402 1.61983388 1.62227488 1.62470706
 1.62713049 1.62954521 1.63195128 1.63434875 1.63673768 1.63911812
 1.64149012 1.64385374 1.64620901 1.64855601 1.65089476 1.65322534
 1.65554777 1.65786213 1.66016844 1.66246677 1.66475715 1.66703964
 1.66931429 1.67158114 1.67384023 1.67609162 1.67833534 1.68057145
 1.68279999 1.68502101 1.68723454 1.68944064 1.69163934 1.69383068
 1.69601472 1.6981915  1.70036105 1.70252342 1.70467865 1.70682678
 1.70896785 1.7111019  1.71322898 1.71534911 1.71746235 1.71956873
 1.72166829 1.72376107 1.7258471  1.72792643 1.72999909 1.73206512
 1.73412456 1.73617745 1.73822381 1.74026369 1.74229712 1.74432414
```

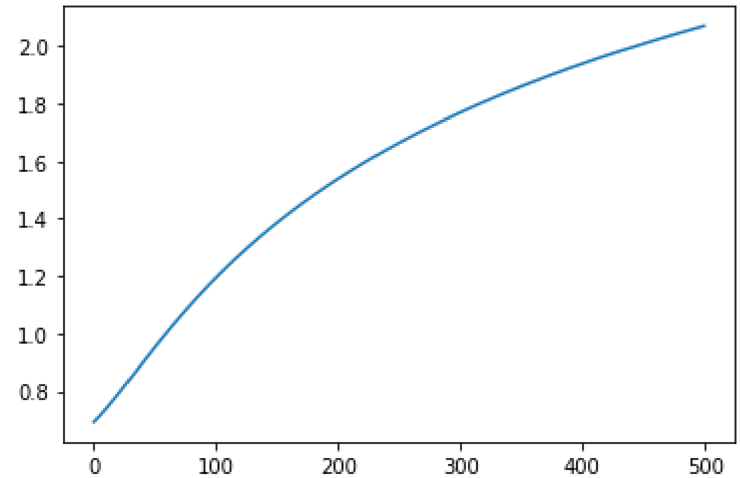
```
 1.74634478 1.74835909 1.75036708 1.75236881 1.7543643  1.75635358
 1.7583367  1.76031368 1.76228456 1.76424937 1.76620814 1.76816091
 1.7701077  1.77204856 1.77398351 1.77591258 1.77783581 1.77975323
 1.78166486 1.78357074 1.7854709  1.78736537 1.78925418 1.79113735
 1.79301492 1.79488692 1.79675338 1.79861432 1.80046977 1.80231977
 1.80416434 1.8060035  1.8078373  1.80966574 1.81148887 1.81330671
 1.81511928 1.81692662 1.81872874 1.82052568 1.82231747 1.82410412
 1.82588566 1.82766213 1.82943354 1.83119992 1.83296129 1.83471769
 1.83646913 1.83821565 1.83995725 1.84169398 1.84342585 1.84515289
 1.84687511 1.84859255 1.85030523 1.85201317 1.8537164  1.85541493
 1.85710879 1.858798   1.86048259 1.86216258 1.86383798 1.86550883
 1.86717514 1.86883694 1.87049424 1.87214707 1.87379545 1.8754394
 1.87707895 1.8787141  1.88034489 1.88197133 1.88359345 1.88521126
 1.88682478 1.88843404 1.89003906 1.89163984 1.89323643 1.89482882
 1.89641705 1.89800113 1.89958108 1.90115692 1.90272867 1.90429635
 1.90585997 1.90741956 1.90897513 1.91052671 1.9120743  1.91361793
 1.91515761 1.91669337 1.91822522 1.91975318 1.92127726 1.92279748
 1.92431387 1.92582643 1.92733519 1.92884016 1.93034136 1.9318388
 1.9333325  1.93482248 1.93630876 1.93779134 1.93927025 1.9407455
 1.94221712 1.9436851  1.94514948 1.94661026 1.94806746 1.9495211
 1.95097119 1.95241775 1.95386079 1.95530033 1.95673638 1.95816896
 1.95959808 1.96102376 1.962446   1.96386484 1.96528027 1.96669232
 1.968101   1.96950632 1.9709083  1.97230695 1.97370228 1.97509432
 1.97648306 1.97786853 1.97925075 1.98062971 1.98200544 1.98337796
 1.98474726 1.98611338 1.98747631 1.98883607 1.99019268 1.99154615
 1.99289649 1.99424372 1.99558784 1.99692887 1.99826682 1.99960171
 2.00093354 2.00226233 2.0035881  2.00491084 2.00623058 2.00754733
 2.0088611  2.0101719  2.01147974 2.01278463 2.01408659 2.01538563
 2.01668176 2.01797499 2.01926533 2.0205528  2.0218374  2.02311914
 2.02439804 2.02567411 2.02694736 2.0282178  2.02948544 2.03075029
 2.03201237 2.03327168 2.03452823 2.03578203 2.0370331  2.03828145
 2.03952708 2.04077001 2.04201024 2.04324779 2.04448267 2.04571489
 2.04694445 2.04817137 2.04939565 2.05061731 2.05183636 2.05305281
 2.05426666 2.05547793 2.05668662 2.05789275 2.05909632 2.06029735
 2.06149584 2.0626918  2.06388525 2.06507619 2.06626462 2.06745057
 2.06863404 2.06981503]
```

In [40]:
```
print(predict(cl, th, vdf[final_words].values))
```

```
[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, 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, 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 [41]:
```
plt.plot(cost)
```

Out[41]: [<matplotlib.lines.Line2D at 0x7f99f0c65fd0>]



In [ ]: