Rishab Kedia

Kevin Liu

Jeffrey Chen

Data Science Project 2 Write-Up

In the sections below, we describe how our team used logistic regression, single-decision tree, and random-forest trees models to classify movie reviews according to their sentiment.

Logistic Regression:

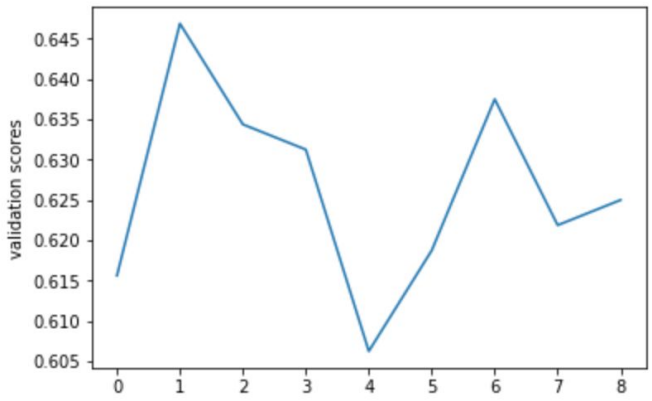
We trained a basic logistic regression model to classify the sentiment of the reviews, and our logistic regression model began with an accuracy of 0.8125. We experimented with the tolerance level, and found that an order of magnitude of 10^-11 seemed to be best. We made the inverse of the regularization strength factor C smaller, so that there was more regularization to reduce overfitting. A C value of 100 seemed to be the best, and we ended with an accuracy of 0.822.

Next, we tried to retain only the features that seemed to be contributing the most towards representing the sentiment of the reviews. We filtered out the outliers by first calculating the mean number of times each word shows up in each column, and then calculating the standard deviation of each of the columns from the mean of all values. The mean turned out to be extremely small at 0.0164 (a lot of the words don't show up that many times throughout all of the documents, so it was expected), but the standard deviation was rather large compared to the mean at 0.3706. Therefore, we thought it was fitting to only select words that showed up at least

1 standard deviations more times than the mean. After filtering out low usage words, we retrained the data, and it resulted in a higher score of 0.8555. We didn't think filtering out more words would be beneficial, because there was the potential that we would be removing some useful data, and our data would be skewed in that case.

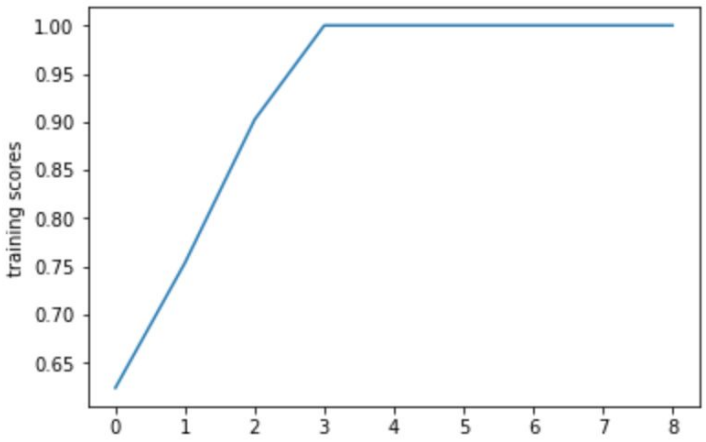
Single-Decision Tree:

Our single decision tree model had an accuracy of 0.628 on the validation set, but had an accuracy of 1.0 on the training set. These results show that the decision tree was overfitting because it completely reduced the error in the training set and was not able to generalize well. One strategy that we had to reduce the overfitting was to limit the depth of the decision tree to prevent the tree from continuing classification until all items in the training set were perfectly classified. To figure out what the maximum depth of the tree should be, we plotted data showing validation scores for a wide range of possible depths as shown below:



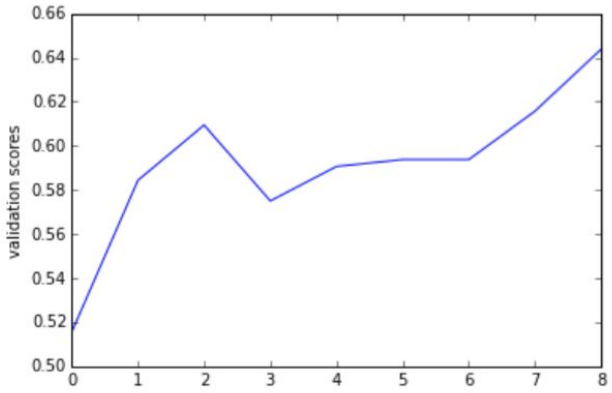
Each mark along the horizontal axis represents one of the possibilities we tried for a potential maximum depth. The entry that corresponded with the peak represented a maximum decision

tree depth of 5, therefore we chose to change our depth limit to 5. We also plotted the training data scores as we changed the maximum depth, as shown below:



As we see can see in the plot, when we allow the tree to have a greater depth, it has a tendency to overfit to the training data, making the training data score increase until it is 1.0, or completely accurate. However, when we stop the depth earlier, the training data has a lower score, meaning that it does not overfit as much, which is why we wanted to limit the maximum depth.

The other parameter that we wanted to change was the maximum number of features included in our decision tree. Primarily, we printed out the features in our model in order of their "importance" to our decision tree, and realized that there were many features that were not very useful. Therefore, we also plotted how the changing the maximum number of features would affect validation scores, as shown below:



Based on this graph, it wasn't obvious what an optimal number of maximum features would be, and if limiting the number of maximum features would even be useful, because even though there was a slight peak in the middle (corresponding to 50 features), the general trend of the graph showed increased validation scores corresponding with an increased amount of features.

For this reason, we limited the maximum depth to 5 and did not limit our number of features, and improved our validation score to 0.656, which was slight improvement over our original decision tree model.

We observed that we were not able to improve the score of the validation test set by that much, and that our model still tends to overfit to the training set. We believe this to be because single-decision trees are very prone to overfitting because unless they are stopped early or "pruned", they tend to classify the training set as perfectly as possible which makes them sacrifice generalizability. Decision trees are particularly prone to overfitting to this data set because it has a lot of irrelevant features. In our data set, the decision tree might make decisions on how to classify movie reviews based on words that have nothing to do with sentiment. The use of the specific feature may work on the training set, but is very unlikely to generalize well.

Since our training set has so many irrelevant features (many words are not pertinent to classification), the single decision tree is very prone to overfitting.

Random-Forest Trees:

Our random forest tree originally had an accuracy of 0.719. We increased the number of trees and found that it helped increase our accuracy dramatically. Even with just small increases, the accuracy jumped significantly, as variance was decreased across more samples. This is in line with our expectations as increasing the number of trees essentially increases the number of decision samples that we are making. We also looked to increase the maximum depth. Deeper trees seemed to do better, as it could reduce bias. The last one that we thought we needed to change was the max number of features. The accuracy vs number of features curve was relatively concave, and there appeared to be a maximum, so we took that value, and combined it with the rest of the parameters, and arrived at a classifier that had an accuracy of 0.98.

In [59]:
```
import numpy as np
import pandas as pd
import os
import sklearn
import matplotlib.pyplot as plt
import math
%matplotlib inline
```

Reading Data

In [2]:
```
def segmentWords(s):
    return s.split()

def readFile(fileName):
    # Function for reading file
    # input: filename as string
    # output: contents of file as list containing single words
    contents = []
    f = open(fileName)
    for line in f:
        contents.append(line)
    f.close()
    result = segmentWords('\n'.join(contents))
    return result
```

Create a Dataframe containing the counts of each word in a file

In [3]:
```
d = []

for c in os.listdir("data_training"):
    directory = "data_training/" + c
    for file in os.listdir(directory):
        words = readFile(directory + "/" + file)
        e = {x:words.count(x) for x in words}
        e['__FileID__'] = file
        e['__CLASS__'] = c
        d.append(e)
```

Create a dataframe from d - make sure to fill all the nan values with zeros.

References:

https://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.html (https://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.html)
https://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.fillna.html (https://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.fillna.html)

In [4]:
```
dataframe = pd.DataFrame(data=d)
dataframe = dataframe.fillna(value=0)
```

In [5]:
```
dataframe['__CLASS__'].head()
```

Out[5]:
```
0    neg
1    neg
2    neg
3    neg
4    neg
Name: __CLASS__, dtype: object
```

Split data into training and validation set

- Sample 80% of your dataframe to be the training data
- Let the remaining 20% be the validation data (you can filter out the indicies of the original dataframe that weren't selected for the training data)

References:

https://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.sample.html (https://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.sample.html) https://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.drop.html (https://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.drop.html)

In [6]:
```
training = dataframe.sample(frac=0.8, random_state=5)
training = training.drop('__FileID__', 1)
indices = training.index
remaining = dataframe.drop(indices)
remaining = remaining.drop('__FileID__', 1)
```

- Split the dataframe for both training and validation data into x and y dataframes - where y contains the labels and x contains the words

References:

https://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.drop.html (https://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.drop.html)

In [43]:
```
y_training = training['__CLASS__']
x_training = training.drop('__CLASS__', 1)
y_validation = remaining['__CLASS__']
x_validation = remaining.drop('__CLASS__', 1)

x_training.head()
```

Out[43]:

		earth	goodies	if	ripley	suspend	they	white			...	zukovsky	zundel	zurg's	zweibel	zwick	zwick's	zwigoff's	zycie	zycie'	|
1510	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
209	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
425	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
982	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
619	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0

5 rows × 45671 columns

Logistic Regression

Basic Logistic Regression

- Use sklearn's linear_model.LogisticRegression() to create your model.
- Fit the data and labels with your model.
- Score your model with the same data and labels.

References:

http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html (http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html)

In [8]:
```
from sklearn import linear_model
log_regression_model = sklearn.linear_model.LogisticRegression()
log_regression_model.fit(x_training, y_training)
log_regression_model.score(x_validation, y_validation)
```

Out[8]:
```
0.8125
```

Changing Parameters

In [9]:
```
logreg = sklearn.linear_model.LogisticRegression(tol=1e-11)
logreg.fit(x_training, y_training)
logreg.score(x_validation, y_validation)
```

Out[9]:
```
0.81562500000000004
```

In [10]:
```
logreg = sklearn.linear_model.LogisticRegression(C = 1e2)
logreg.fit(x_training, y_training)
logreg.score(x_validation, y_validation)
```

Out[10]:
```
0.82187500000000002
```

In [11]:
```
means = []
for row in x_training:
    means.append(x_training[row].mean())
```

In [12]:
```
means = np.asarray(means)
means
```

Out[12]:
```
array([ 0.00078125,  0.00078125,  0.00078125, ...,  0.        ,
        0.        ,  0.        ])
```

In [13]:
```
means.size
```

Out[13]:
```
45671
```

In [14]:
```
m = means.mean()
```

In [15]:
```
std = means.std()
```

In [16]:
```
#has to be +1 sd or more
x_validation= x_validation.drop([row for row in x_training if x_training[row].mean() - m < std], axis=1)
x_training = x_training.drop([row for row in x_training if x_training[row].mean() - m < std], axis=1)
```

In [17]:
```
x_training.head()
```

Out[17]:

	!	"	(	)	*	,	-	--	.	:	...	who	why	will	with	work	world	would	years	you	your
1510	0	0	7	7	0	29	2	0	28	0	...	2	0	2	5	1	1	0	0	1	0
209	0	4	2	2	0	33	0	0	26	1	...	2	1	1	7	0	0	2	0	4	0
425	0	0	7	7	4	47	0	5	36	5	...	1	2	0	5	0	0	1	0	0	0
982	0	0	1	1	0	41	0	0	32	1	...	4	0	1	5	0	1	4	0	1	0
619	1	6	10	12	2	86	0	2	52	1	...	3	1	0	9	1	0	2	1	1	1

5 rows × 202 columns

In [18]:
```
logreg = sklearn.linear_model.LogisticRegression(tol=1e-11)
logreg.fit(x_training, y_training)
```

Out[18]:
```
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
          intercept_scaling=1, max_iter=100, multi_class='ovr',
          penalty='l2', random_state=None, solver='liblinear', tol=1e-11,
          verbose=0)
```

In [19]:
```
x_training.shape
```

Out[19]:
```
(1280, 202)
```

In [20]:
```
y_training.shape
```

Out[20]:
```
(1280,)
```

In [21]:
```
x_validation.shape
```

Out[21]:
```
(320, 202)
```

In [22]:
```
y_validation.shape
```

Out[22]:
```
(320,)
```

In [23]:
```
logreg.score(x_training, y_training)
```

Out[23]:
```
0.85546875
```

Feature Selection

- In the backward stepsize selection method, you can remove coefficients and the corresponding x columns, where the coefficient is more than a particular amount away from the mean - you can choose how far from the mean is reasonable.

References:

https://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.html# (https://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.html#)
https://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.sample.html (https://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.sample.html) https://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.drop.html (https://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.drop.html) http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html (http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html) https://docs.scipy.org/doc/numpy-1.13.0/reference/generated/numpy.where.html (https://docs.scipy.org/doc/numpy-1.13.0/reference/generated/numpy.where.html) https://docs.scipy.org/doc/numpy-1.13.0/reference/generated/numpy.std.html (https://docs.scipy.org/doc/numpy-1.13.0/reference/generated/numpy.std.html) https://docs.scipy.org/doc/numpy-1.13.0/reference/generated/numpy.mean.html (https://docs.scipy.org/doc/numpy-1.13.0/reference/generated/numpy.mean.html)

In [ ]:
```

```

How did you select which features to remove? Why did that reduce overfitting?

In [ ]:
```

```

Single Decision Tree

Basic Decision Tree

- Initialize your model as a decision tree with sklearn.
- Fit the data and labels to the model.

References:

http://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html (http://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html)

In [25]:
```
from sklearn.tree import DecisionTreeClassifier
decision_tree_model = sklearn.tree.DecisionTreeClassifier()
decision_tree_model.fit(x_training, y_training)
decision_tree_model.score(x_validation, y_validation)
```

Out[25]: 0.60624999999999996

Changing Parameters

- To test out which value is optimal for a particular parameter, you can either loop through various values or look into sklearn.model_selection.GridSearchCV

References:

http://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html (http://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html) http://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html (http://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html)

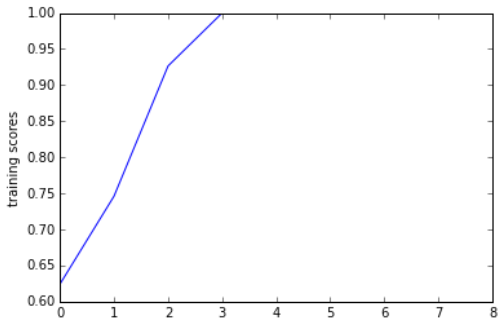
In [26]:
```
from sklearn.tree import DecisionTreeClassifier
classifier = DecisionTreeClassifier()

classifier.fit(x_training, y_training)
importances = classifier.feature_importances_
std = np.std(classifier.feature_importances_,
             axis=0)
indices = np.argsort(importances)[::-1]

# Print the feature ranking
print("Feature ranking:")

for f in range(x_training.shape[1]):
    print("%d. feature %d (%f)" % (f + 1, indices[f], importances[indices[f]]))
```

Feature ranking:
1. feature 22 (0.078566)
2. feature 31 (0.077213)
3. feature 5 (0.025619)
4. feature 150 (0.022612)
5. feature 99 (0.021964)
6. feature 76 (0.019963)
7. feature 43 (0.019391)
8. feature 113 (0.017427)
9. feature 28 (0.016429)
10. feature 119 (0.016126)
11. feature 171 (0.015759)
12. feature 121 (0.015287)
13. feature 11 (0.014918)
14. feature 91 (0.014752)
15. feature 174 (0.014749)
16. feature 108 (0.013872)
17. feature 145 (0.013755)
18. feature 166 (0.012495)
19. feature 44 (0.012495)
20. feature 186 (0.012423)
21. feature 93 (0.012326)
22. feature 60 (0.012246)
23. feature 161 (0.011365)
24. feature 181 (0.011150)
25. feature 68 (0.011092)
26. feature 134 (0.011060)
27. feature 50 (0.010587)
28. feature 156 (0.010462)
29. feature 173 (0.010288)
30. feature 46 (0.009956)
31. feature 200 (0.009922)
32. feature 148 (0.009886)
33. feature 19 (0.009429)
34. feature 12 (0.009197)
35. feature 9 (0.009115)
36. feature 79 (0.008802)
37. feature 1 (0.008793)
38. feature 185 (0.008730)
39. feature 89 (0.008727)
40. feature 132 (0.008522)
41. feature 2 (0.008511)
42. feature 20 (0.008478)
43. feature 64 (0.008377)
44. feature 81 (0.008343)
45. feature 88 (0.008145)
46. feature 56 (0.008004)
47. feature 120 (0.007955)
48. feature 197 (0.007939)
49. feature 6 (0.007934)
50. feature 133 (0.007879)
51. feature 35 (0.007848)
52. feature 194 (0.006990)
53. feature 168 (0.006768)
54. feature 77 (0.006710)
55. feature 47 (0.006667)
56. feature 167 (0.006563)
57. feature 182 (0.006492)
58. feature 152 (0.006201)
59. feature 78 (0.006013)
60. feature 75 (0.005869)
61. feature 199 (0.005854)
62. feature 123 (0.005667)
63. feature 126 (0.005530)
64. feature 158 (0.005377)
65. feature 25 (0.005208)
66. feature 172 (0.005093)
67. feature 179 (0.005093)
68. feature 128 (0.004831)
69. feature 62 (0.004815)
70. feature 74 (0.004688)
71. feature 10 (0.004688)
72. feature 160 (0.004654)
73. feature 127 (0.004640)
74. feature 86 (0.004551)
75. feature 30 (0.004511)
76. feature 94 (0.004491)
77. feature 42 (0.004464)
78. feature 0 (0.004464)
79. feature 23 (0.004464)
80. feature 187 (0.004167)
81. feature 135 (0.004087)
82. feature 170 (0.003750)
83. feature 138 (0.003516)
84. feature 122 (0.003509)
85. feature 175 (0.003295)
86. feature 54 (0.003061)
87. feature 144 (0.003045)
88. feature 45 (0.003043)

89. feature 65 (0.003030)
90. feature 176 (0.003005)
91. feature 32 (0.002969)
92. feature 188 (0.002951)
93. feature 37 (0.002918)
94. feature 189 (0.002909)
95. feature 83 (0.002902)
96. feature 155 (0.002887)
97. feature 151 (0.002742)
98. feature 69 (0.002734)
99. feature 137 (0.002730)
100. feature 95 (0.002679)
101. feature 149 (0.002679)
102. feature 115 (0.002604)
103. feature 13 (0.002344)
104. feature 14 (0.002344)
105. feature 130 (0.002344)
106. feature 67 (0.002083)
107. feature 3 (0.002083)
108. feature 15 (0.002083)
109. feature 198 (0.002083)
110. feature 124 (0.002083)
111. feature 139 (0.002083)
112. feature 105 (0.002083)
113. feature 110 (0.001902)
114. feature 40 (0.001702)
115. feature 52 (0.001563)
116. feature 125 (0.001492)
117. feature 7 (0.001296)
118. feature 49 (0.000978)
119. feature 57 (0.000000)
120. feature 33 (0.000000)
121. feature 53 (0.000000)
122. feature 27 (0.000000)
123. feature 34 (0.000000)
124. feature 29 (0.000000)
125. feature 58 (0.000000)
126. feature 55 (0.000000)
127. feature 36 (0.000000)
128. feature 4 (0.000000)
129. feature 18 (0.000000)
130. feature 38 (0.000000)
131. feature 17 (0.000000)
132. feature 8 (0.000000)
133. feature 39 (0.000000)
134. feature 51 (0.000000)
135. feature 41 (0.000000)
136. feature 26 (0.000000)
137. feature 24 (0.000000)
138. feature 48 (0.000000)
139. feature 21 (0.000000)
140. feature 16 (0.000000)
141. feature 59 (0.000000)
142. feature 201 (0.000000)
143. feature 61 (0.000000)
144. feature 63 (0.000000)
145. feature 136 (0.000000)
146. feature 140 (0.000000)
147. feature 141 (0.000000)
148. feature 142 (0.000000)
149. feature 143 (0.000000)
150. feature 146 (0.000000)
151. feature 147 (0.000000)
152. feature 153 (0.000000)
153. feature 154 (0.000000)
154. feature 157 (0.000000)
155. feature 159 (0.000000)
156. feature 162 (0.000000)
157. feature 163 (0.000000)
158. feature 164 (0.000000)
159. feature 165 (0.000000)
160. feature 169 (0.000000)
161. feature 177 (0.000000)
162. feature 178 (0.000000)
163. feature 180 (0.000000)
164. feature 183 (0.000000)
165. feature 184 (0.000000)
166. feature 190 (0.000000)
167. feature 191 (0.000000)
168. feature 192 (0.000000)
169. feature 193 (0.000000)
170. feature 195 (0.000000)
171. feature 196 (0.000000)
172. feature 131 (0.000000)
173. feature 129 (0.000000)
174. feature 118 (0.000000)
175. feature 96 (0.000000)
176. feature 66 (0.000000)
177. feature 70 (0.000000)

```
178. feature 71 (0.000000)
179. feature 72 (0.000000)
180. feature 73 (0.000000)
181. feature 80 (0.000000)
182. feature 82 (0.000000)
183. feature 84 (0.000000)
184. feature 85 (0.000000)
185. feature 87 (0.000000)
186. feature 90 (0.000000)
187. feature 92 (0.000000)
188. feature 97 (0.000000)
189. feature 117 (0.000000)
190. feature 98 (0.000000)
191. feature 101 (0.000000)
192. feature 102 (0.000000)
193. feature 103 (0.000000)
194. feature 104 (0.000000)
195. feature 106 (0.000000)
196. feature 107 (0.000000)
197. feature 109 (0.000000)
198. feature 111 (0.000000)
199. feature 112 (0.000000)
200. feature 114 (0.000000)
201. feature 116 (0.000000)
202. feature 100 (0.000000)
```

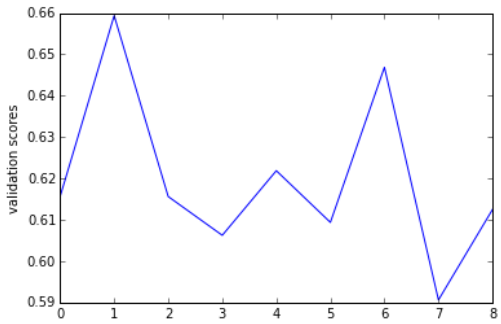
In [27]:
```
plt_training_scores = [];
plt_val_scores = [];

for i in [1, 5, 10, 50, 100, 500, 1000, 5000, 10000]:
    new_tree_model = sklearn.tree.DecisionTreeClassifier(max_depth=i)
    new_tree_model.fit(x_training, y_training)
    plt_training_scores.append(new_tree_model.score(x_training, y_training))
    plt_val_scores.append(new_tree_model.score(x_validation,y_validation))

plt.plot(plt_training_scores)
plt.ylabel('training scores')
plt.show()
```



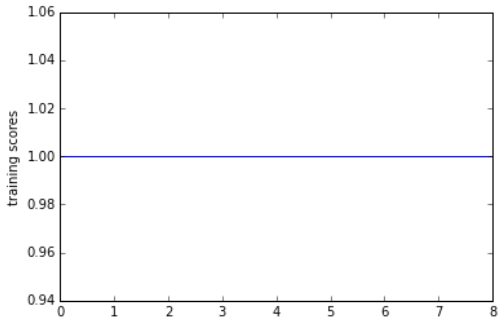
How did you choose which parameters to change and what value to give to them? Feel free to show a plot.

(The parameters are changed at the end of this section). I changed max_depth to 5, and max_features to 10,000. This increased the score from 0.628 to 0.656.

In [28]:
```
plt.plot(plt_val_scores)
plt.ylabel('validation scores')
plt.show()
```



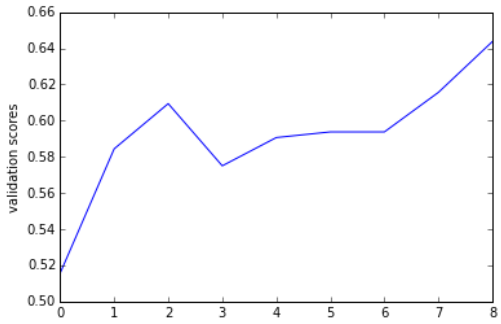
In [45]:
```
plt_training_scores = [];
plt_val_scores = [];

for i in [1, 10, 50, 100, 500, 1000, 5000, 10000, 40000]:
    new_tree_model = sklearn.tree.DecisionTreeClassifier(max_features=i)
    new_tree_model.fit(x_training, y_training)
    plt_training_scores.append(new_tree_model.score(x_training, y_training))
    plt_val_scores.append(new_tree_model.score(x_validation,y_validation))

plt.plot(plt_training_scores)
plt.ylabel('training scores')
plt.show()
```



Why is a single decision tree so prone to overfitting?

Decisions Trees are prone to overfitting because unless you stop them before completion, they will always perfectly categorize all the elements in the test set, meaning that they do not generalize well and overfit.

In [46]:
```
plt.plot(plt_val_scores)
plt.ylabel('validation scores')
plt.show()
```



In [47]:
```
improved_tree_model = sklearn.tree.DecisionTreeClassifier(max_depth=5, max_features=100)
improved_tree_model.fit(x_training, y_training)
improved_tree_model.score(x_validation, y_validation)
```

Out[47]: 0.59062499999999996

Random Forest Classifier

Basic Random Forest

- Use sklearn's ensemble.RandomForestClassifier() to create your model.
- Fit the data and labels with your model.
- Score your model with the same data and labels.

References:

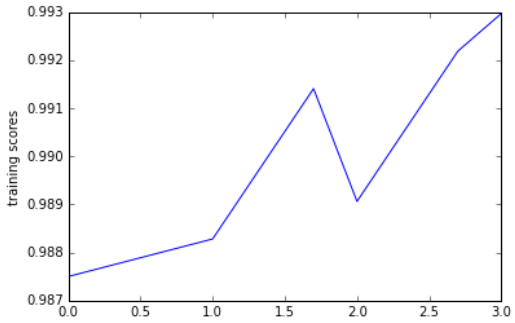
http://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html (http://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html)

In [48]:
```
from sklearn.ensemble import RandomForestClassifier
random_forest = RandomForestClassifier()
random_forest.fit(x_training, y_training)
random_forest.score(x_validation, y_validation)
```

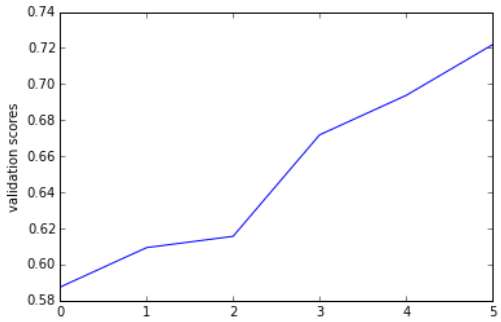
Out[48]: 0.71875

Changing Parameters

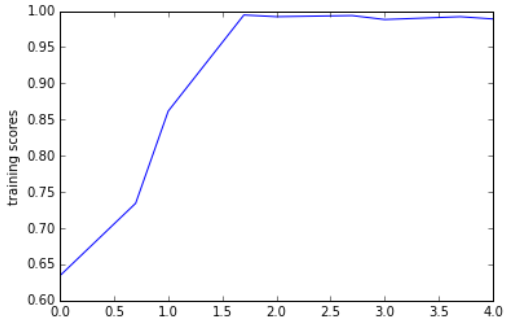
In [63]:
```
plt_training_scores = [];
plt_val_scores = [];
features = [1, 10, 50, 100, 500, 1000]
for i in features:
    new_tree_model = RandomForestClassifier(max_features=i)
    new_tree_model.fit(x_training, y_training)
    plt_training_scores.append(new_tree_model.score(x_training, y_training))
    plt_val_scores.append(new_tree_model.score(x_validation,y_validation))

plt.plot([math.log10(x) for x in features], plt_training_scores)
plt.ylabel('training scores')
plt.show()
```



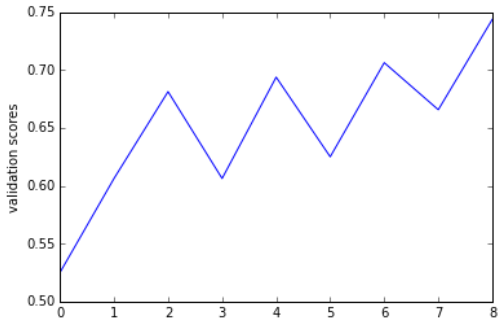
In [50]:
```
plt.plot(plt_val_scores)
plt.ylabel('validation scores')
plt.show()
```



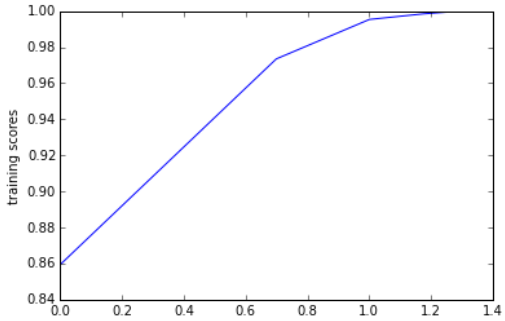
In [61]:
```
plt_training_scores = [];
plt_val_scores = [];
depth = [1, 5, 10, 50, 100, 500, 1000, 5000, 10000]
for i in depth:
    new_tree_model = RandomForestClassifier(max_depth=i)
    new_tree_model.fit(x_training, y_training)
    plt_training_scores.append(new_tree_model.score(x_training, y_training))
    plt_val_scores.append(new_tree_model.score(x_validation,y_validation))

plt.plot([math.log10(x) for x in depth], plt_training_scores)
plt.ylabel('training scores')
plt.show()
```



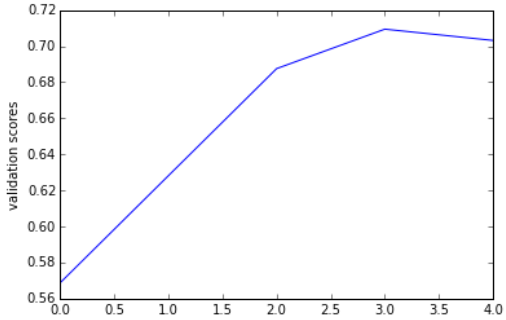
In [52]:
```
plt.plot(plt_val_scores)
plt.ylabel('validation scores')
plt.show()
```



In [67]:
```
plt_training_scores = [];
plt_val_scores = [];
estimators = [1, 5, 10, 15, 20]
for i in estimators:
    new_tree_model = RandomForestClassifier(n_estimators=i)
    new_tree_model.fit(x_training, y_training)
    plt_training_scores.append(new_tree_model.score(x_training, y_training))
    plt_val_scores.append(new_tree_model.score(x_validation,y_validation))

plt.plot([math.log10(x) for x in estimators], plt_training_scores)
plt.ylabel('training scores')
plt.show()
```



In [68]:
```
plt.plot(plt_val_scores)
plt.ylabel('validation scores')
plt.show()
```



What parameters did you choose to change and why?

In [70]:
```
new_tree_model = RandomForestClassifier(n_estimators=15, max_depth = 500, max_features=50)
new_tree_model.fit(x_training, y_training)
new_tree_model.score(x_training, y_training)
```

Out[70]: 0.99765625000000002

How does a random forest classifier prevent overfitting better than a single decision tree?

In [ ]:

In [ ]: