CS168 Spring Assignment 2

SUNet ID(s):	bgrd	tannerj	mbolivas
Name(s):	Benjamin Doughty	Tanner Jensen	Micah Olivas

By turning in this assignment, I agree by the Stanford honor code and declare that all of this is my own work.

Part 1

(a) **Part 1 Code**

```
def jaccard_similarity(mat):
    """
        inputs:
            -mat : a 2 row matrix
        returns:
            Jaccard similarity of the 2 rows
    """
    return mat.min(axis=0).sum() / mat.max(axis=0).sum()

def L2_similarity(a, b):
    """
        inputs:
            -a : a csr sparse vector
            -b : another csr sparse vector
        returns:
            the L2 similarity of the two sparse vectors
    """
    return -np.linalg.norm(a - b)

def cosine_similarity(a, b):
    """
        inputs:
            -a : a csr sparse vector
            -b : another csr sparse vector
        returns:
            the cosine similarity of the two sparse vectors
    """
    return (a @ b.T) / (np.linalg.norm(a)*np.linalg.norm(b))

dat = pd.read_csv('data50.csv', sep = ',', names = ["articleId", "wordId", "Count"]
groups = pd.read_csv('p2_data/groups.csv', names = ['Name'])
```

```
labels = pd.read_csv('p2_data/label.csv', names = ['Label'])
#word_mat = sparse.csr_matrix((dat.Count, (dat.articleId, dat.wordId))).todense()

# Sparse matrix implementation is slow, so store full matrix for now
word_mat = sparse.csr_matrix((dat.Count, (dat.articleId, dat.wordId))).todense()

similarity_matrix = np.zeros((3,len(groups),len(groups)))
for i,j in itertools.combinations_with_replacement(groups.index, 2):
    # row number in file is 0-indexed but labels and articles are 1-indexed so we a
    articles_i = labels[labels.Label == i + 1].index + 1
    articles_j = labels[labels.Label == j + 1].index + 1
    pairwise_results = np.zeros((3, len(articles_i)*len(articles_j)))
    k = 0
    for a,b in itertools.product(articles_i, articles_j):
        pairwise_results[0, k] = jaccard_similarity(word_mat[[a, b]])
        pairwise_results[1, k] = L2_similarity(word_mat[a], word_mat[b])
        pairwise_results[2, k] = cosine_similarity(word_mat[a], word_mat[b])
        k += 1
    similarity_matrix[:,i,j] = similarity_matrix[:,j,i] = pairwise_results.mean(axi
```

(b) We plot below the average pairwise article similarities between the 20 news groups using Jaccard, L2, and cosine similarity metrics in figure 1 below.

(c) Our criteria for determining a good distance metric is the assumption that articles from the same news group should be more similar to each other than articles from 2 separate news groups. Also, articles from two news groups in the same semantic hierarchy (i.e. two articles in computer-related groups) should be more similar than articles from different semantic hierarchies (i.e a computer article and a recreation article). From these assumptions, we'd expect a good distance metric heatmap to have a strong diagonal with a modular structure where semantic hierarchical clusters are apparent.

Looking at the 3 metrics we plotted, it is clear that L2 euclidean distance is a poor metric as it makes the news groups appear far more similar than they should intuitively be, masking the true differences and structure between news groups. Jaccard and cosine similarity heatmaps are comparably good: jaccard has a stronger diagonal but cosine similarity preserves more of the modular hierarchical structure; however, jaccard similarity also has a major weakness because it performs poorly when the two documents are of different lengths. Large gaps between the min and max of the counts of words caused by a difference in document length strongly bias the Jaccard similarity. Cosine similarity does not suffer from this problem because, Geometrically, the cosine

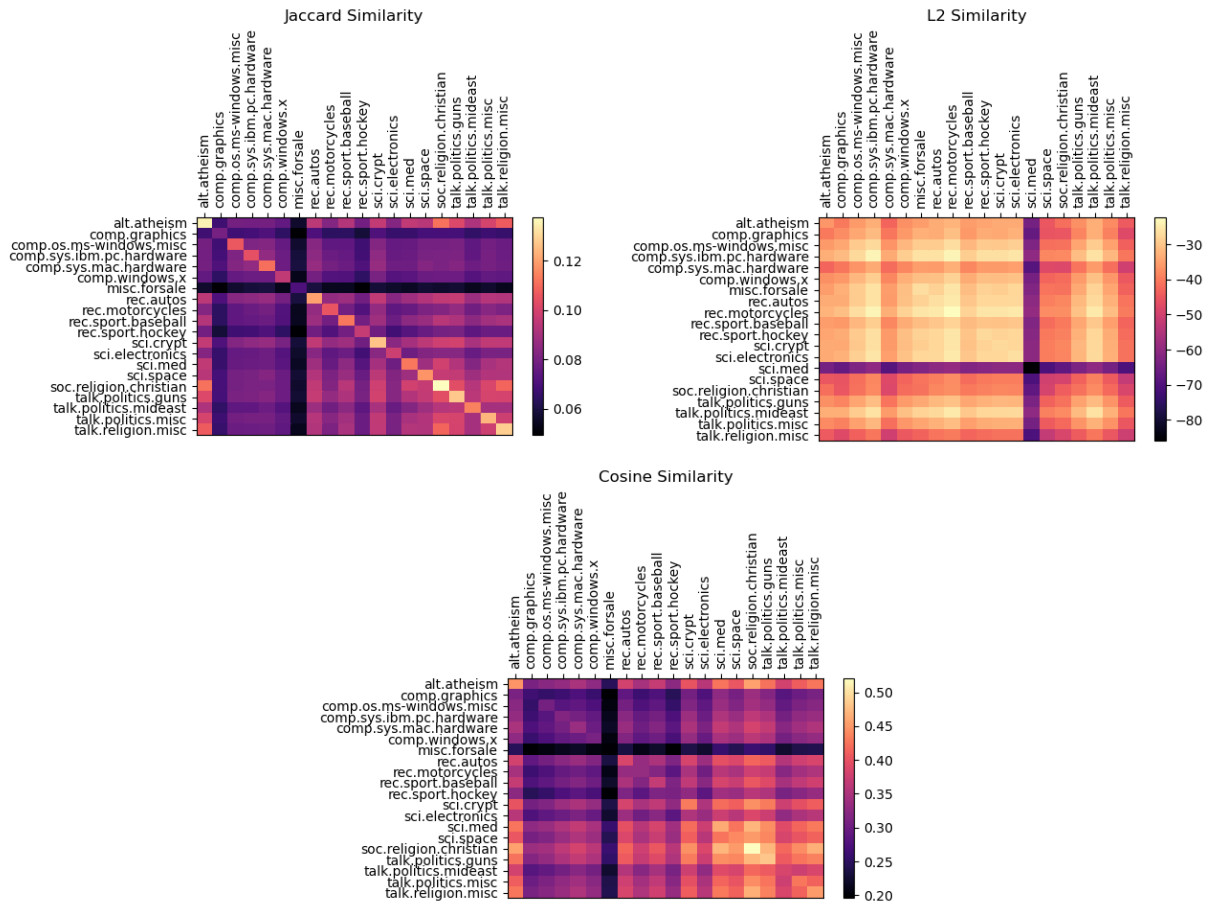

Figure 1: Average similarity of articles from 20 news groups using different similarity metrics

similarity is the angle between the vectors in n-dimensional space. This means it depends much less on the magnitude of words (how many times each word appears in the document) and rather the cumulative direction of the vector. The cosine similarity, therefore, is a well-suited metric for comparing documents.

Outside of the diagonal which is expected to be most similar, the most similar news groups were *soc.religion.christian* and *talk.religion.misc* which makes sense given the language of both articles will probably revolve around similar religious talking points and evangelical phrases. Somewhat shockingly, *alt.atheism* is also very similar to *soc.religion.christian*, though this makes sense when considering atheist channels will often reference christian and other religious beliefs and talking points while making their case against them.

Part 2

(a)
```
def brute_force_search(query, search_space, metric=cosine_similarity, self=0):
    similarity_vec = np.apply_along_axis(lambda row: metric(row, query),
                                         1,
                                         search_space)
    similarity_vec[0,self] = np.NINF
    max_idx = np.nanargmax(similarity_vec)
    return max_idx

def build_classification_matrix(data_mat, groups=groups, labels=labels):
    classification_matrix = np.zeros((len(groups),len(groups)))

    for i in range(len(labels)):
        true_label = labels.loc[i,'Label']
        doc = data_mat[i]
        most_sim_doc = brute_force_search(doc, data_mat, self=i)
        most_sim_label = labels.loc[most_sim_doc,'Label']
        classification_matrix[true_label-1,most_sim_label-1] += 1

    return classification_matrix

def compute_classification_accuracy(classification_mat):
    return classification_mat.diagonal().sum() / classification_mat.sum()
```

The heatmap is plotted in figure 2. The average classification accuracy is: 0.456.

(b) In part 1, all the similarity metrics were commutative, i.e. J(S,T) = J(T,S), which means that the similarity matrices were symmetric. However, the quantity that we are computing in (a) above is not commutative, since "number of As with nearest neighbor

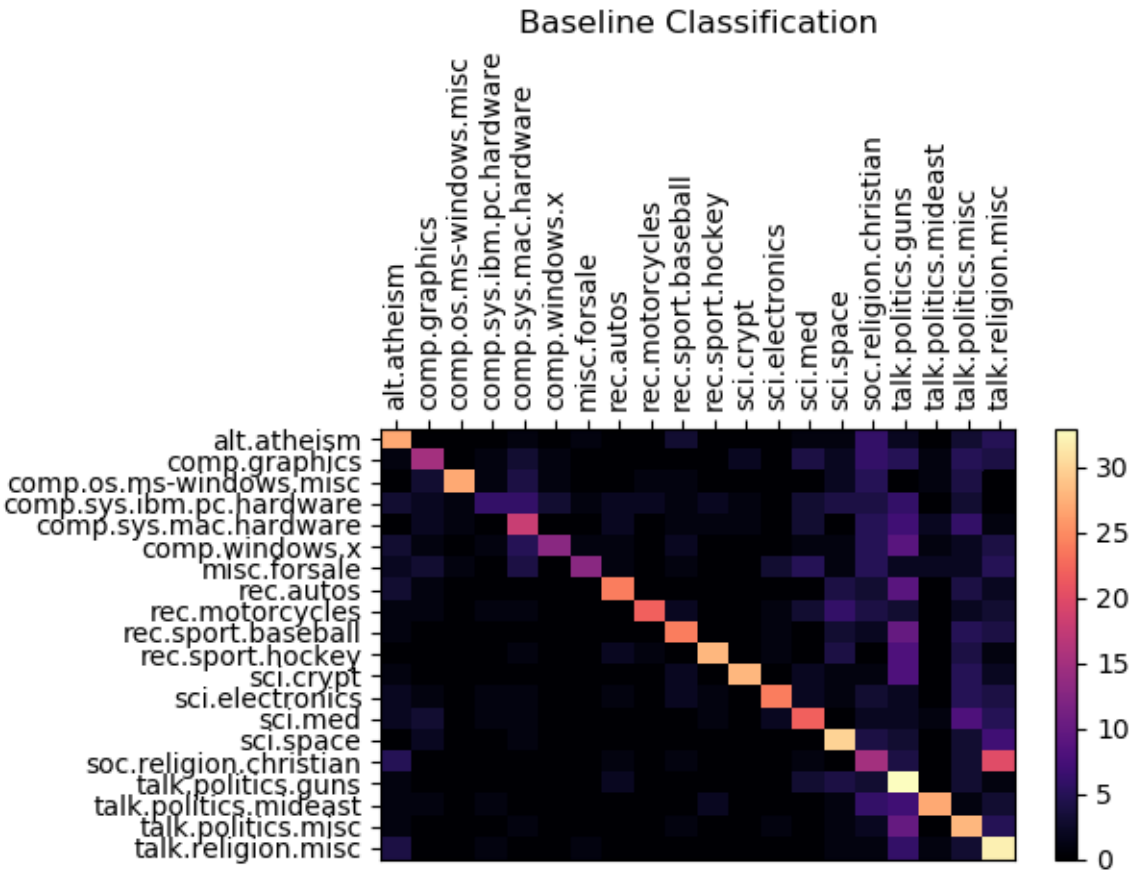

Figure 2: Baseline classification Matrix Heatmap.

in B" does not equal the "number of Bs with nearest neighbor in A" in the general case.

(c)
```
ds = [10, 25, 50, 100, 200, 500, 1000]
k = word_mat.shape[1]

dim_red_classification_matrix = np.zeros((len(ds),len(groups),len(groups)))

for idx, d in enumerate(ds):
    random_mat = np.random.normal(size=(k,d))
    projected_mat = word_mat @ random_mat
    classification_mat = build_classification_matrix(projected_mat)
    dim_red_classification_matrix[idx] = classification_mat
```

The results of the plots for the various values of d are plotted in figure 3.

Dimension d	**Classification Accuracy**
10	0.121
25	0.254
50	0.321
100	0.363
200	0.412
500	0.440
1000	0.452

For $d = 500$ we start to see classification accuracy approaching with the full matrix.

(d) To reduce the dimension of the data requires multiplying a $n \times k$ matrix by a $k \times d$ matrix for a run time of $\mathcal{O}(nkd)$. Classifying an article requires projecting the new article into low-dimensional space ($\mathcal{O}(kd)$) and then computing a dot product of two d-dimensional vectors n times ($\mathcal{O}(nd)$), for a total time complexity of $\mathcal{O}((n+k)d) \approx \mathcal{O}(nd)$.

For the sparse-tweet example, since we know nearly all of the k entries in the bag-of-words vector will be 0 for two tweets, we can only focus on the entries that are non-zero in one tweet. In particular, instead of computing the dot-product by iterating over all indices $\{1, 2, \ldots, k\}$, we can simply loop over the 50 non-zero entries in the first tweet and check if they are in the other (then normalize by the norms, which also only take 50 operations each to compute).

The resulting runtimes for the sparse-tweet example (50 operations per article by n articles) is the same as for the dimensionality reduction method for $d = 50$.

Part 3

(a) $\mathbb{P}(x \text{ and } y \text{ in same bucket}) = (1 - \frac{\theta}{\pi})^d$. In $\mathbb{R}^2$, each vector is normal to a line that divides space into the regions where the dot product is positive and where it is negative,

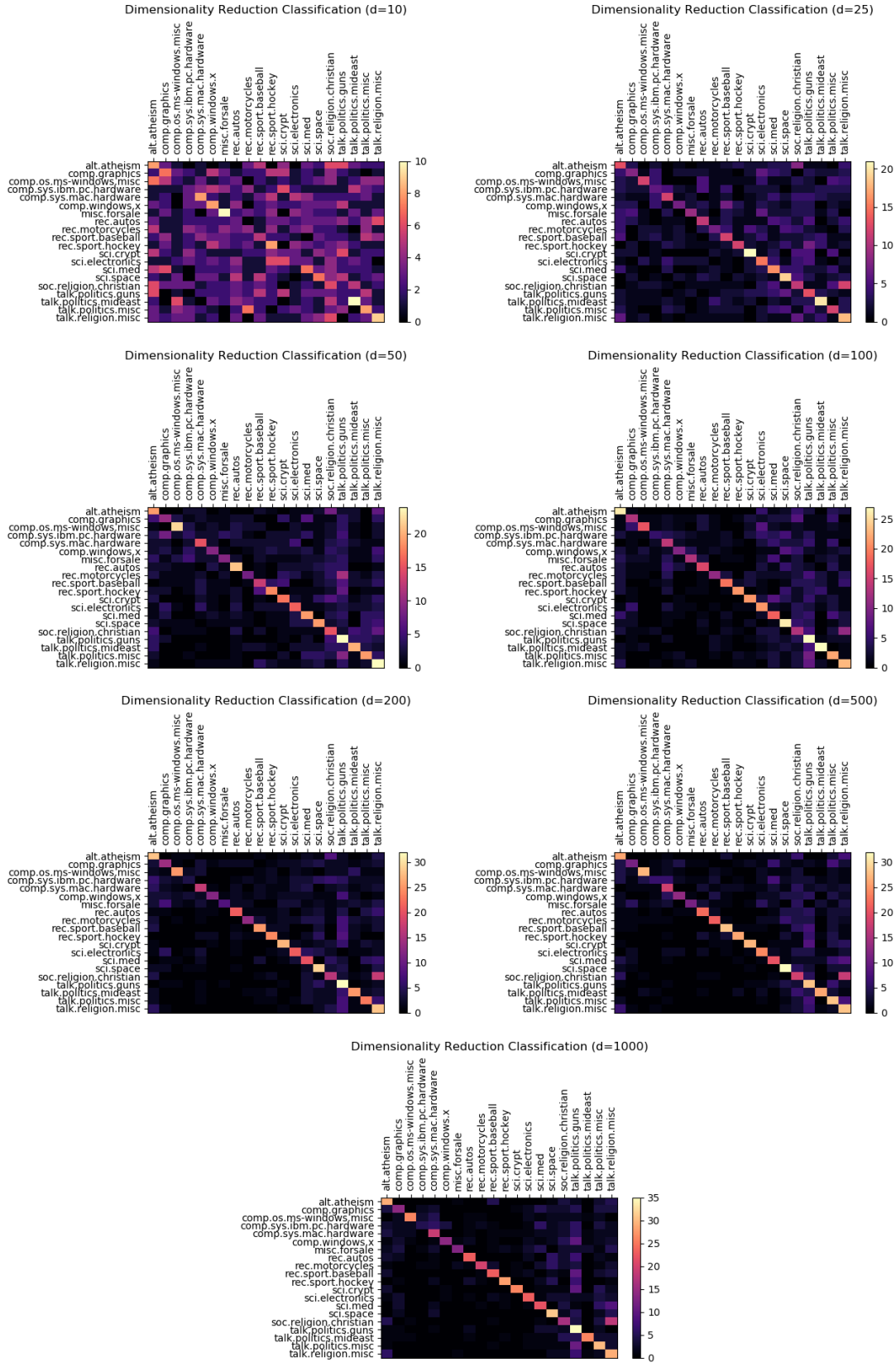

Figure 3: Heatmaps for classification after dimensionality reduction with varying d.

so the probability that the two vectors have opposite signs after projection is the probability that a random vector falls into the region between these two normal lines; the angle between these lines is exactly θ, so the probability that a random vector falls in the region is $\frac{\theta}{\pi}$ (in $\mathbb{R}^k$ the same logic holds, since we can project the random vector onto the plane spanned by x and y and ignore the remaining orthogonal components). If the probability that the signs differ for a given coordinate is $\frac{\theta}{\pi}$, the probability that all d signs are the same is $(1 - \frac{\theta}{\pi})^d$.

(b) The optimal values, given the problem description, are $l \approx 168, d \approx 133$. Briefly, we achieved this value with a grid search over l and d with the time complexity in (2) as the objective function, with the constraint from (1). First, we found that the probability that a random point falls into at least one of the same buckets as a fixed point θ radians away is equal to $1 - (1 - (1 - \frac{\theta}{\pi})^d)^l$. So we can constrain our search for values of l, d where $1 - (1 - (1 - \frac{.1}{\pi})^d)^l > 0.9$. Then, we note that the expected number of items that will overlap is just $N' = N(1 - (1 - (1 - \frac{0.2}{\pi})^d)^l)$. Finally, we note that the time complexity is composed of two terms, a hashing term ($\mathcal{O}(kld)$) and a search term ($\mathcal{O}(kN')$). We can factor out irrelevant k and take $\arg\min_{l,d} ld + N'$ confined to $1 - (1 - (1 - \frac{.1}{\pi})^d)^l > 0.9$. By plotting the objective function, the boundary, and the individual terms that make up the boundary, we can notice a few interesting things (figure 4). First, higher values of l seem to increase the cost while higher values of d decrease it. This is a consequence of the probability expression above (more hash tables means high collision probability, bigger hash tables means smaller collision probability). However, with too low values of l with respect to d, we can see that we fail to capture the target point with high probability. Finally, we note that the optimal point roughly balances the contribution to the cost of both terms, which makes sense as a tradeoff between the two costs. While this optimization makes sense theoretically, in practice space considerations mean we should increase l while decreasing d, since the space required to store a set of (naive) hashtables is linear in l and exponential in d.

(c)
```
def sign_vec_to_int(sign_vec):
    return int(''.join(map(str,(.5*(sign_vec+1)).astype(np.int).tolist()[0])),2)

def compute_hash_val(v, mat):
    sign_vec = np.sign(v @ mat)
    return sign_vec_to_int(sign_vec)

def build_hashtables(dat_mat, d, k, l, n=1000):
    random_mats = [np.random.normal(size=(k,d)) for _ in range(l)]
    hash_tables = [{} for _ in range(l)]
    for j in range(n):
        v = dat_mat[j]
        for i in range(l):
```

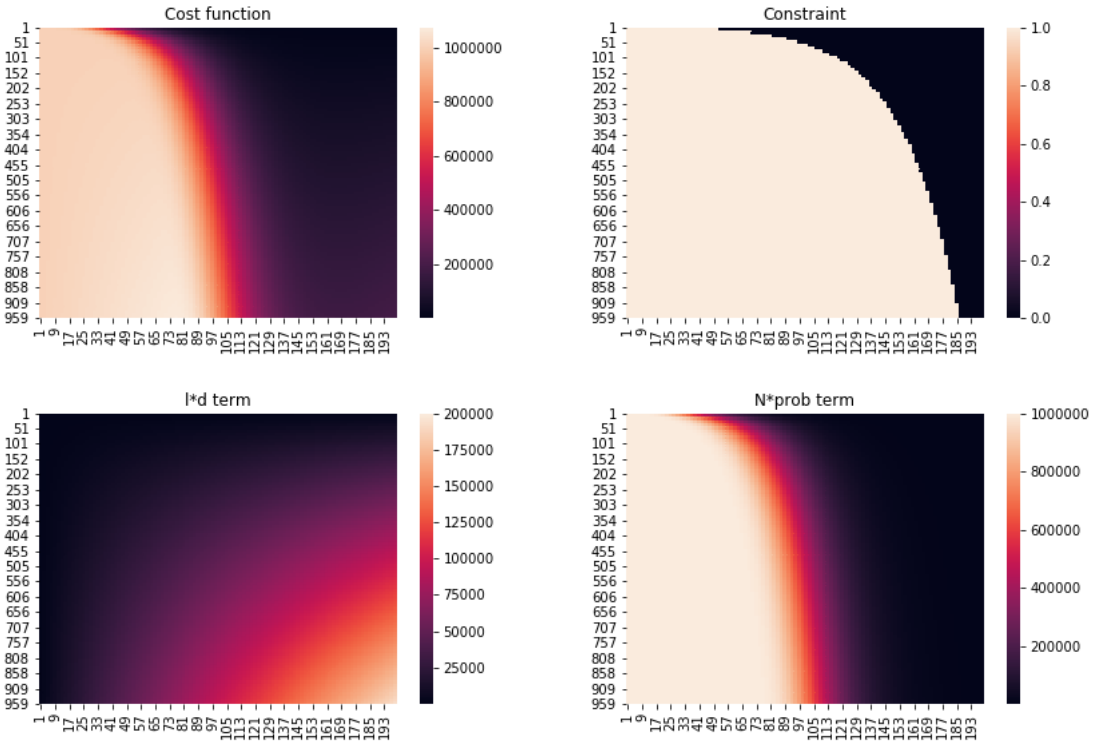

Figure 4: Grid search results for optimization problem.

```
            mat = random_mats[i]
            hash_i = compute_hash_val(v, mat)
            curr_val = hash_tables[i].get(hash_i, [])
            hash_tables[i][hash_i] = curr_val + [j]
    return random_mats, hash_tables

def query(q, mats, hash_tables):
    possible_matches = []
    for i in range(l):
        mat = mats[i]
        hash_i = compute_hash_val(q, mat)
        possible_matches += hash_tables[i].get(hash_i, [])
    return set(possible_matches)

def build_classification_matrix(data_mat, mats, hash_tables, groups, labels):
    classification_matrix = np.zeros((len(groups),len(groups)))

    Sqs = []

    for i in range(len(labels)):
        true_label = labels.loc[i,'Label']
        doc = data_mat[i]
        possible_docs = query(doc, mats, hash_tables)
        possible_docs = list(possible_docs)
        Sqs.append(len(possible_docs)-1)
        subset_data_mat = data_mat[possible_docs]
        subset_labels = labels.loc[possible_docs]
        self_idx = possible_docs.index(i)
        most_sim_doc = brute_force_search(doc, subset_data_mat, self=self_idx)
        most_sim_label = subset_labels.iloc[most_sim_doc,0]
        classification_matrix[true_label-1,most_sim_label-1] += 1

    return classification_matrix, np.mean(Sqs)
```

The results of comparing classification accuracy by the size of the search space is plotted in figure 5. As the size of the search space goes down, so does our accuracy, since we have fewer chances to get it right. However, we can still retain good accuracy and reduced search time with $d \approx 11$ (roughly the inflection point on the graph).

(d) Consisting of a hashing step with run time $\mathcal{O}(kld)$ and a query step with time $\mathcal{O}(kN')$, LSH operates in time $\mathcal{O}(kld + kN') \approx \mathcal{O}(kN')$. Compared to similarity search by dimensional reduction, which has a time complexity of $\mathcal{O}(dN)$, identifying similar vectors by purely locality-sensitive hashing appears to be the less efficient method for similar-

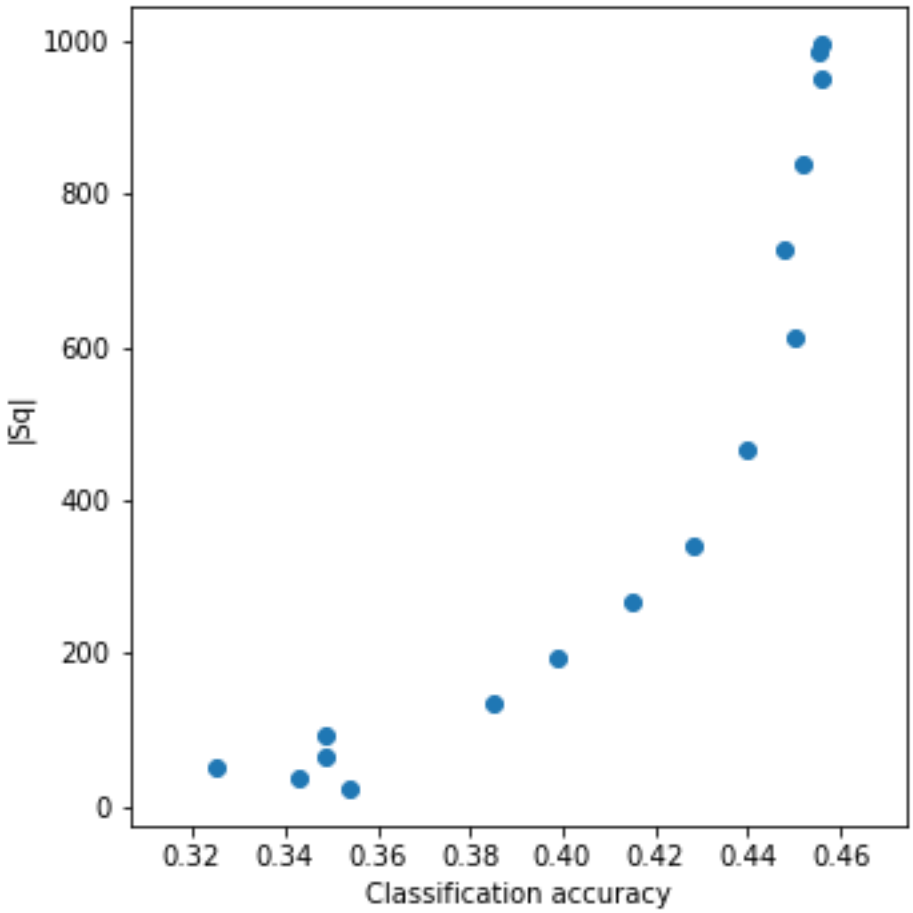

Figure 5: Tradeoff between search-space size and classification accuracy for LSH.

ity approximation applications with a low required specificity (for object x in set X with highest similarity to observation y, $\mathbb{P}(x \text{ and } y \text{ in same bucket}) < 0.9$) and sublogarithmic operating time. The best use cases for either approach are, understandably, those data structures that minimize the values of k and d for LSH and dimensional reduction, respectively.