CMSC 726 Final Project Report: Classifying UFO Sightings

Alex J. Malozemoff

December 19, 2012

Abstract

We investigate the feasibility of classifying UFO sightings in unsupervised and semi-supervised settings. Using data scraped from the National UFO Reporting Center website, we apply both K-means and self-training with SVM wrapper functions. Our results are two-fold. On the negative side, we show that K-means does not effectively cluster the sightings according to our projected labels. On the positive side, we find that self-training is an effective method at classifying sightings; we achieve an accuracy of 43% when using a linear-SVM wrapper function.

1 Introduction

The National UFO Reporting Center (NUFORC) provides a public database[1] of reported UFO sightings. Each report contains the sighting's date, duration, location, shape of object, as well as a description of the sighting. A lot of these reports can be easily explained by natural phenomena. For example, many sightings can be explained as being sightings of the International Space Station (ISS), stars, planets, etc. However, such classification requires much manual labor, correlating UFO reports with, for instance, the expected location of the ISS at that time. We attempt to automate this process by applying both unsupervised and semi-supervised machine learning techniques, as follows.[2]

Unsupervised Learning. Without external labels, the classification problem is clearly of the unsupervised variety: we are given a set of features (extracted from the UFO reports) and wish to cluster the data, hopefully in such a way that each cluster represents a specific type of sighting (such as "satellite", "planet/star", etc.). We apply the K-means algorithm to the dataset and evaluate the results.

Semi-supervised Learning. Not all of the data is in fact unlabeled. NUFORC occasionally provides notes in each report, detailing whether they believe the sighting was in fact a satellite, a star, a hoax, etc. These data provide important details about several of the sightings which we utilize for a semi-supervised approach to the problem. The goal is thus, given the few reports that contain labels, to cluster the unlabeled data. We implement a self-training semi-supervised learning algorithm to classify the data, using a support vector machine (SVM) as the underlying supervised learning algorithm.

[1] http://www.nuforc.org/webrreports.html

[2] All algorithms/methodologies come from the CMSC726 readings and slides provided unless explicitly stated. All implementations are my own.

Applying these algorithms to the dataset, we find that the unsupervised approach does not produce clusters which match the expected labels. However, our semi-supervised learning approach performs remarkably well, achieving an accuracy of around 43%, even though only 4.4% of the dataset contains initially labeled data.

The rest of the paper proceeds as follows. In Section 2, we detail how we construct our dataset. Section 3 details the application of unsupervised learning techniques to our dataset, whereas Section 4 describes the use of semi-supervised techniques. Finally, we conclude in Section 5.

2 Constructing the Dataset

In this section, we describe our methodology for constructing the dataset.

The first step was to construct a feature vector for each UFO report documented by NUFORC. We scraped the NUFORC website, downloading every UFO report between 11/2012 and 01/2000; this dataset comprises around 59,000 reports. From these reports, we processed them to extract the time of the sighting, the duration of the sighting, the location of the sighting (in latitude and longitude), the object's shape, and any other pertinent features to be described below. In the process, we threw out any sightings that contained "invalid" entries, such as bogus durations (e.g., values such as "?") and other such errors. Also, to simplify matters we focused exclusively on UFO sightings in the contiguous United States. This left around 32,000 sightings in our final dataset.

Most of the feature-extraction process was straightforward, except for extracting the latitude and longitude of each sighting. We now describe this process. Given as input the town and state of the sighting, we determined the FIPS code[3] of this location, and used the datasets from `https://www.census.gov/geo/www/cob/pl2000.html` to convert the FIPS code to a latitude-longitude pair. As the census' town-to-FIPS dataset does not include unincorporated areas, for any missing towns we utilized the Google Maps API to determine that town's geolocation.[4] (The reason we did not use the Google Maps API from the get-go was because they limit the number of queries that can be made in a given 24-hour period to 2,500, and thus geolocating all 59,000 sightings would take an unreasonable amount of time.)

For extracting features from the sighting description, we used a basic "bag-of-words" approach. We first calculate the count of all words found in the text, and did a manual scan for "interesting" words, such as colors (e.g., "blue", "silver", etc.), and characteristics of the sighting (e.g., "blink" for a potentially blinking UFO, "abduct" for a claimed abduction, etc.). In the end, we are left with 31,509 data points in the contiguous United States, with each datapoint containing 37 features; see Table 1 for a list of the features and their (un-normalized) means.

To aid in understanding the dataset, Figure 1 plots each sighting by its latitude and longitude and colored by its shape. This gives a basic outline of how sightings are distributed across the United States between January 2000 and November 2012. For an easier-to-parse presentation of the above data, we constructed a time-lapsed video of all sightings between 01/2000 and 08/2012, which can be viewed at `http://www.cs.umd.edu/~amaloz/ufo/movie1.avi`.

Of the available data points, several of them are in fact labeled. The reports on the NUFORC website occasionally include comments listing whether the sighting is a planet, satellite, etc. We extracted these comments from the reports and manually investigated them to determine appropri-

[3]For information on FIPS codes, see `http://quickfacts.census.gov/qfd/meta/long_fips.htm`

[4]We make use of the googlemaps Python library to interface with the Google Maps API; see `http://pypi.python.org/pypi/googlemaps/`.

feature	**mean**	**feature**	**mean**
daytime	964.722873	duration	812.850328
lat	38.465973	lng	-95.206323
white	0.294709	red	0.718652
orange	0.179790	blue	0.135898
green	0.121584	silver	0.046558
gold	0.017360	yellow	0.082453
gray	0.020026	black	0.074772
blink	0.114158	abduct	0.004189
circle	0.094513	disk	0.052429
triangle	0.104986	chevron	0.013679
rectangle	0.017677	fireball	0.078327
formation	0.031039	light	0.223079
changing	0.026342	unknown	0.074677
other	0.068139	oval	0.047161
diamond	0.014599	sphere	0.066552
flash	0.018376	teardrop	0.010473
cigar	0.024437	egg	0.009521
cross	0.003364	cylinder	0.016757
cone	0.003872		

Table 1: Un-normalized features and their means across the entire dataset.

label	advertising-lights	planet/star	contrail	satellite	hoax	aircraft	mystery	balloon
count	64	391	62	334	198	285	28	25

Table 2: Table of possible sighting labels and their counts in the dataset.

ate labels, eventually settling on eight possible labels; see Table 2. This process left us with 1,387 labeled data points. See Figure 2 for a plot of these labels. Even from this small sample size, we can see some interesting features. Note that, in general, planet/star and satellite sightings occur throughout the U.S., which is to be expected (as one can presumably see such objects anywhere on Earth). However, note how advertising lights, and to a lesser extent, hoaxes, predominate around major city centers. This again matches our intuition.

Before proceeding to the learning algorithms, we did two additional steps in pre-processing the dataset. First, we did *feature pruning*, where we removed any features that were either too infrequent or too common. We used a cutoff point of 1%; that is, if a feature occurred in less than one percent of the examples or more than ninety-nine percent of the examples, we removed that feature from consideration. This reduced the number of considered features to 33 (removing the 'abduct', 'egg', 'cross', and 'cone' features). Second, we normalized the feature set so that every feature falls between -1 and 1. This prevents continuous features with large ranges (such as latitude or longitude) from overwhelming binary features in the learning process.

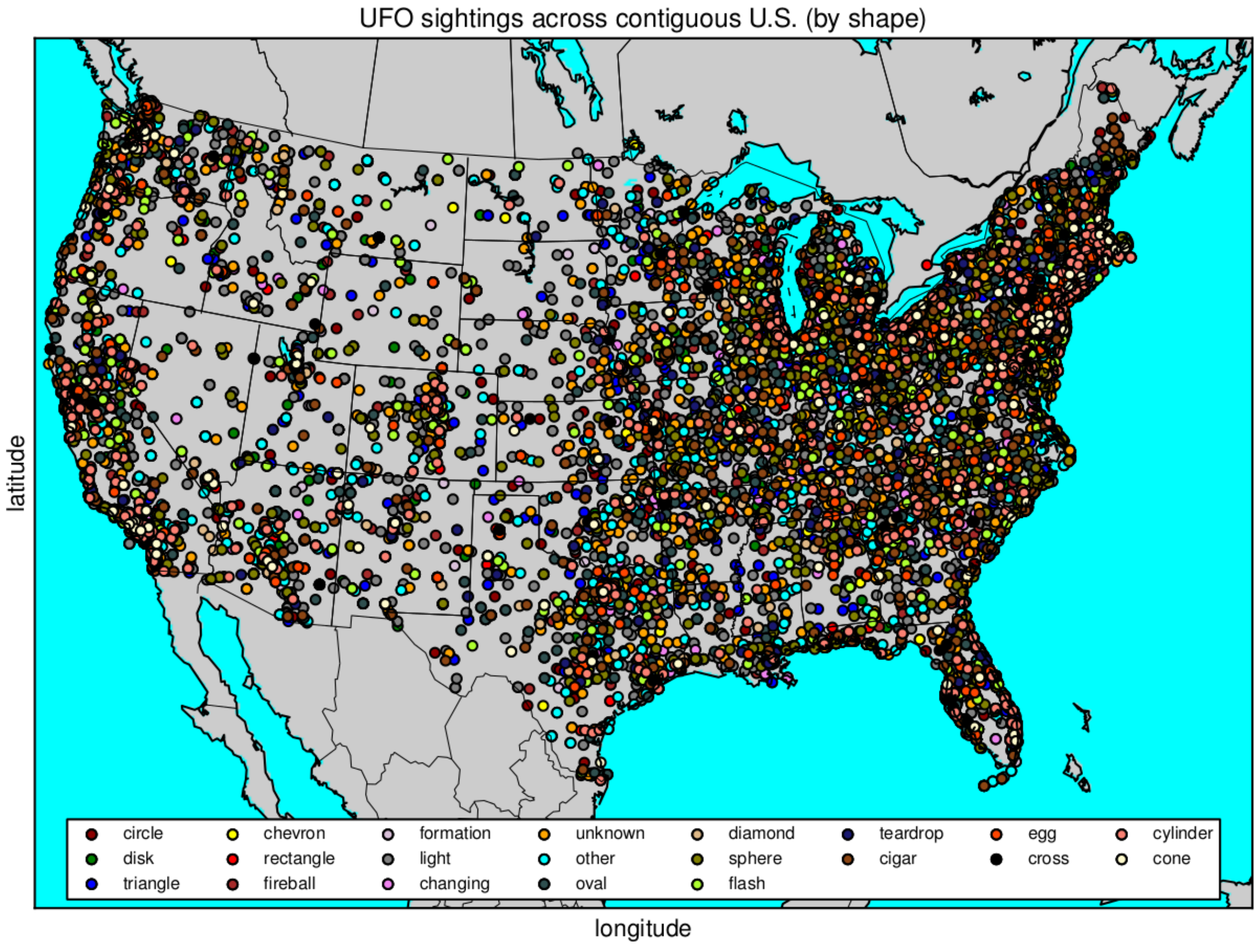

Figure 1: Plot of latitude/longitude of UFO sightings across the contiguous United States, colored by shape.

3 Unsupervised Learning

We now detail our investigations applying the K-means unsupervised learning algorithm to our dataset. See Appendix A for the associated source code. Our implementation is a straight-forward adaption of K-means using the furthest-first heuristic to pick our initial cluster points. We apply this algorithm to our dataset, setting $K = 8$, which is the number of labels we extracted above (cf. Table 2). Figure 3 plots each sighting colored by the associated cluster. It is hard to gather anything informative from this figure, besides seeing that the 8th cluster appears to dominate. Thus, to properly evaluate the performance of our K-means implementation on the dataset, we use the set of labeled data we *do* have as an "evaluation set". Thus, we evaluate the K-means algorithm by measuring the label distribution of the correctly labeled examples within a given cluster returned by K-means, using the entropy as our measure.[5] Using such a method on the clusters output by K-means gives a score of 1.64. This value is not too useful without a comparison point, so we re-run

[5]This method was suggested by Prof. Corrada Bravo.

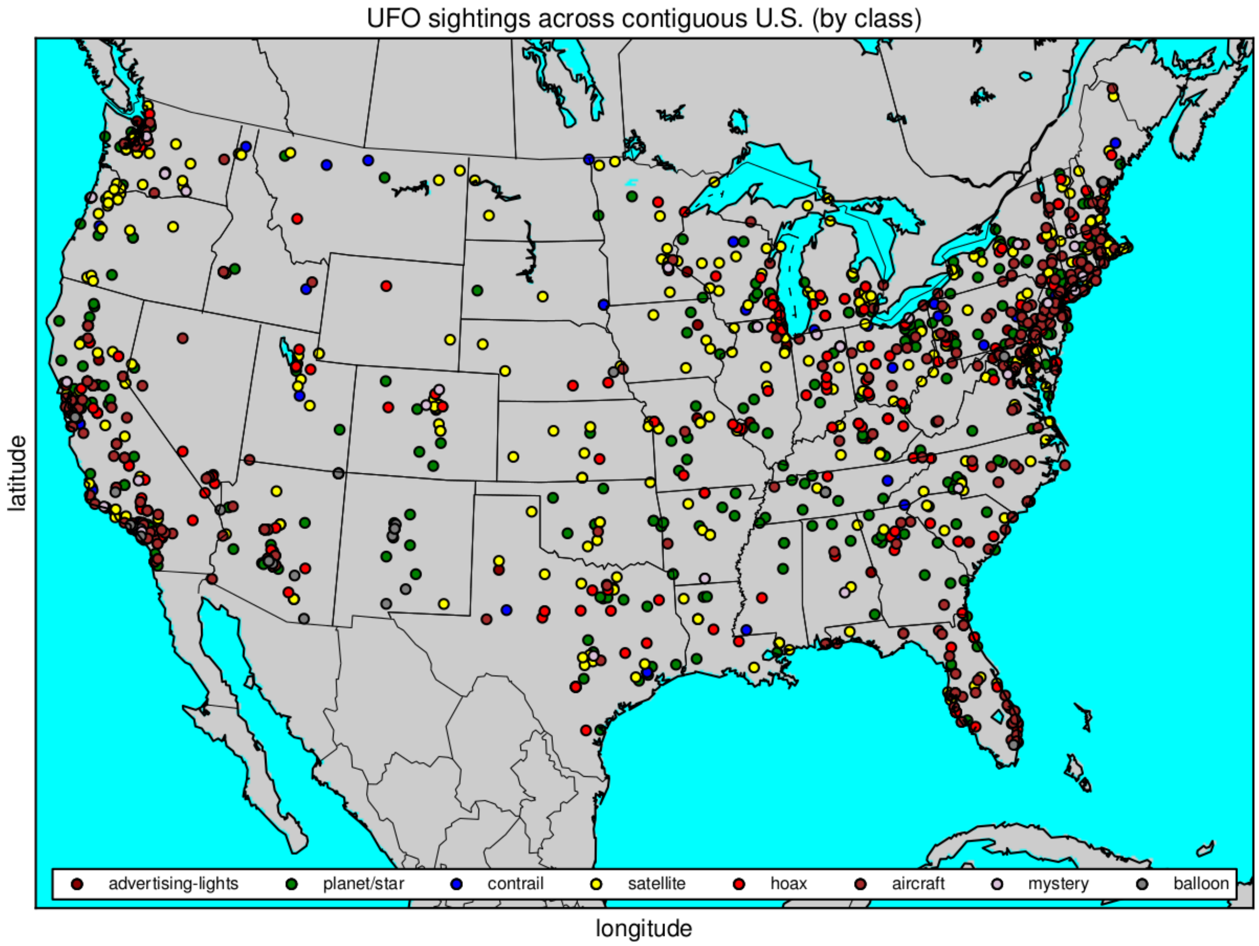

Figure 2: Plot of latitude/longitude of UFO sightings across the contiguous United States, colored by label.

K-means, but this time for only a *single* iteration. The intuition behind this is that by running for only a single iteration, we expect a larger score as the algorithm generally needs to iterate to effectively cluster sightings (that is, the objective function will be higher and thus the resulting score should be higher as well). However, we see that using a single iteration gives a score of 1.51! Further testing reinforced these results. This seems to suggest that an unsupervised approach is *not* good at clustering according to the labels in Table 2, as we achieve a better score by not even iterating.

Let's look at this more closely to see why. For a given cluster, we take the mean across all features of those examples to see if we can identify any patterns. And in fact, we can. Many clusters are formed around a single feature. For example, cluster 2 in Figure 3 in fact just clusters all "blue" UFOs. Similarly, cluster 3 is composed of all "white" UFOs. Meanwhile, our labels do not follow such strict adherence to a single feature. Thus, without *external* guidance, K-means appears to focus in on single features to differentiate clusters, which doesn't match how the labels are actually distributed. This seems to suggest that we need to utilize the few labels we *do* have to

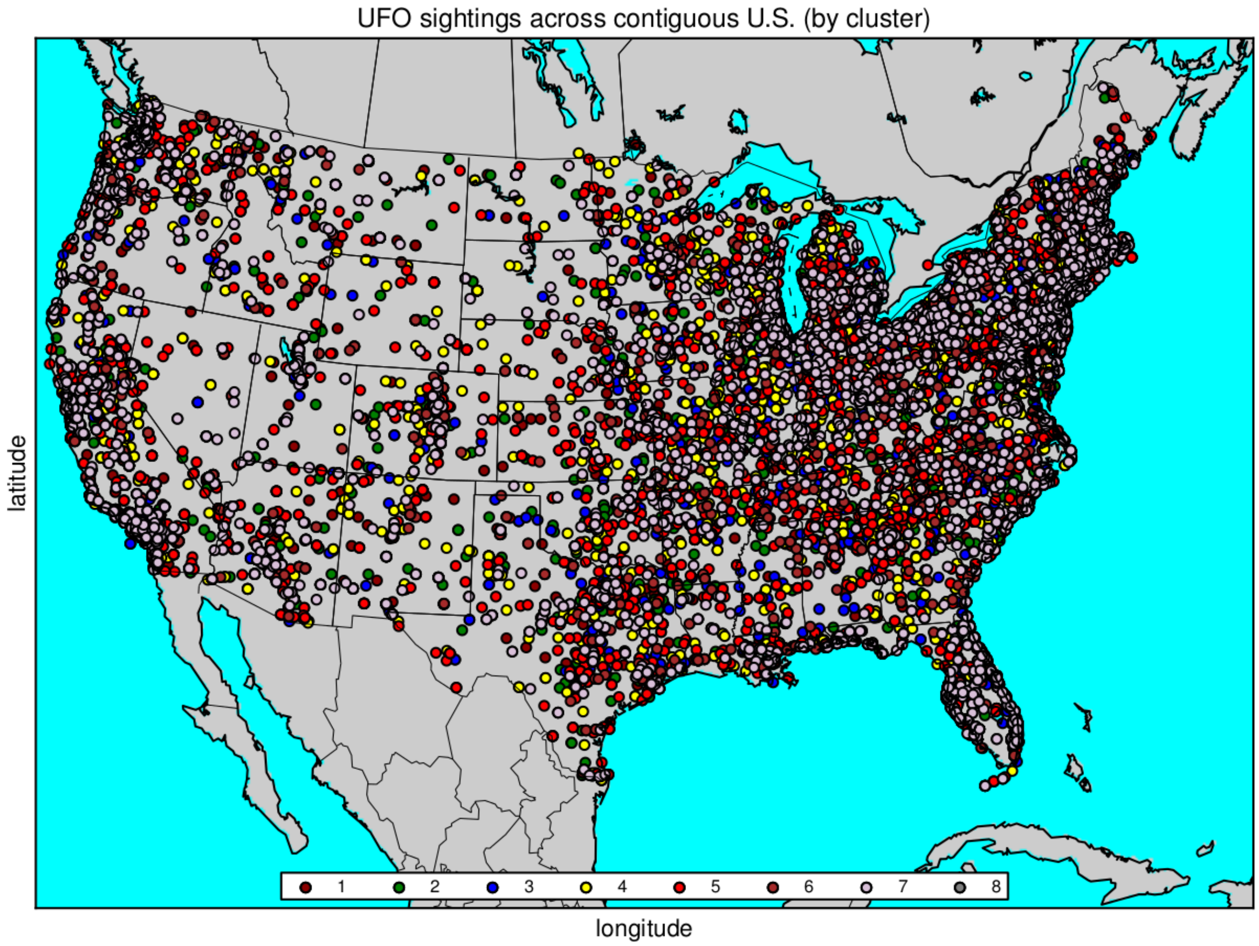

Figure 3: Plot of latitude/longitude of UFO sightings across the contiguous United States, colored by cluster.

better cluster the data, which leads to our semi-supervised approach, described below.

4 Semi-supervised Learning

We implement a standard self-training semi-supervised learning algorithm, and apply it to the UFO dataset. See Appendix B for the associated source code. The algorithm works by using the labeled data as our training set. Upon training a classifier on the labeled data (where the classifier can be chosen as an external parameter), we use the classifier to predict labels for our unlabeled data. For each example, the classifier outputs confidence probabilities for each label; that is, the probability that a given example is classified using a given label. We select the `size` most confident examples and label them accordingly, where here `size` represents the number of examples to classify in any given round (another external parameter to the algorithm). The larger the value of `size`, the fewer number of iterations are required to train the entire dataset. After `size` examples are classified, we repeat the whole process, using the newly labeled items as the training set for our classifier.

We utilize the `sklearn.svm` module[6] to provide the underlying SVM classifiers (specifically, we experiment with the `sklearn.svm.SVC` object, a C-Support Vector Classifier) and construct our code so that these objects can be plugged in easily, to aid experimenting with different SVM methods. For evaluation, we implement 10-fold cross-validation. That is, we split the labeled data into ten "chunks", and in each iteration we remove one of these "chunks" from our labeled data and train using what's left, using the held-out "chunk" as as our evaluation set. We use the standard accuracy measure (i.e., the number of correct matches divided by the total number of items in the evaluation set) to rate the performance.

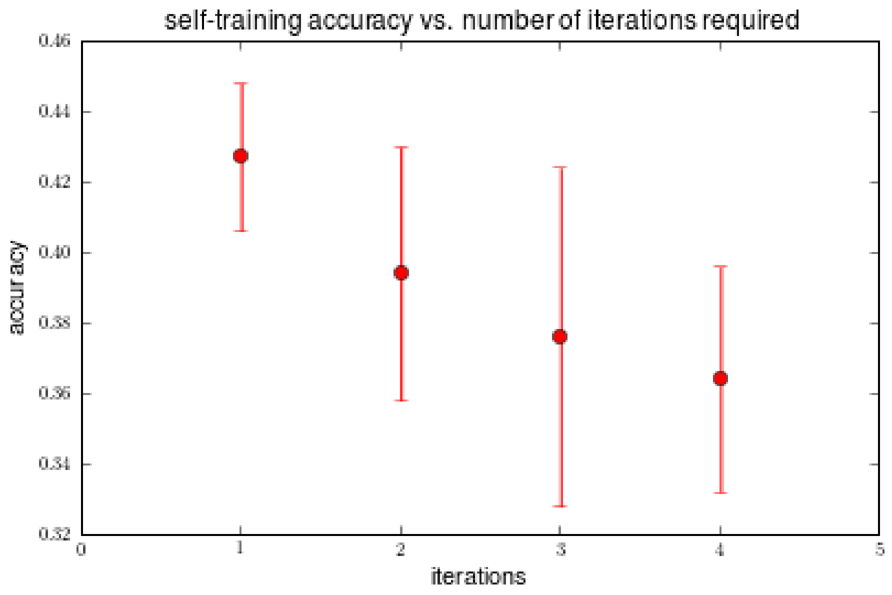

Figure 4: Accuracy of our linear-svm-based self-training classifier different numbers of iterations required to classify the whole dataset.

Figure 4 shows the performance of a linear SVM ($C = 1$) using different numbers of iterations in our self-training algorithm (the larger the number of iterations, the smaller the value of size described above). We see here that as the number of iterations gets larger, our performance *decreases*! However, this can be easily explained. One problem with self-training algorithms is that they can propagate mistakes, as in subsequent iterations whatever the algorithm classified before is now treated as ground truth. We see such an effect here; as the number of iterations made by the self-training algorithm *increases*, performance *decreases*. Thus, from here on out we only use a single iteration, and thus set size equal to the number of items needing to be classified (around 30,000). (This is also good in the sense that the larger the number of iterations, the longer the running time. Thus, by limiting the number of iterations to one we also gain in much more efficient algorithms.)

Figure 5 shows the performance of our self-training algorithm on four different kernel SVMs: linear, RBF, degree-3 polynomial, and degree-4 polynomial. One thing we immediately notice is the large variation in performance across the folds of cross-validation. This makes sense when considering that our labeled data is rather small, and thus the evaluation set in each fold only consists of around 130 examples. However, one surprising thing is how well these methods work.

[6] http://scikit-learn.org/stable/

For our best algorithm (linear-svm-based self-training with $C = 6$) we get an average accuracy of 43%. While at first glance this may not seem great, recall that this is classifying across eight categories. Thus, a *random* guess would only give us a 13% chance of success, so we are doing *much* better than random. Similarly, just picking the majority label ("planet/star" according to Table 2) would give us only 28%. Also recall the great variability in these reports. Our feature set is composed of basic elements from the sightings reports, as well as a simple bag-of-words feature extraction method. Even with such simple methods, we get quite good performance!

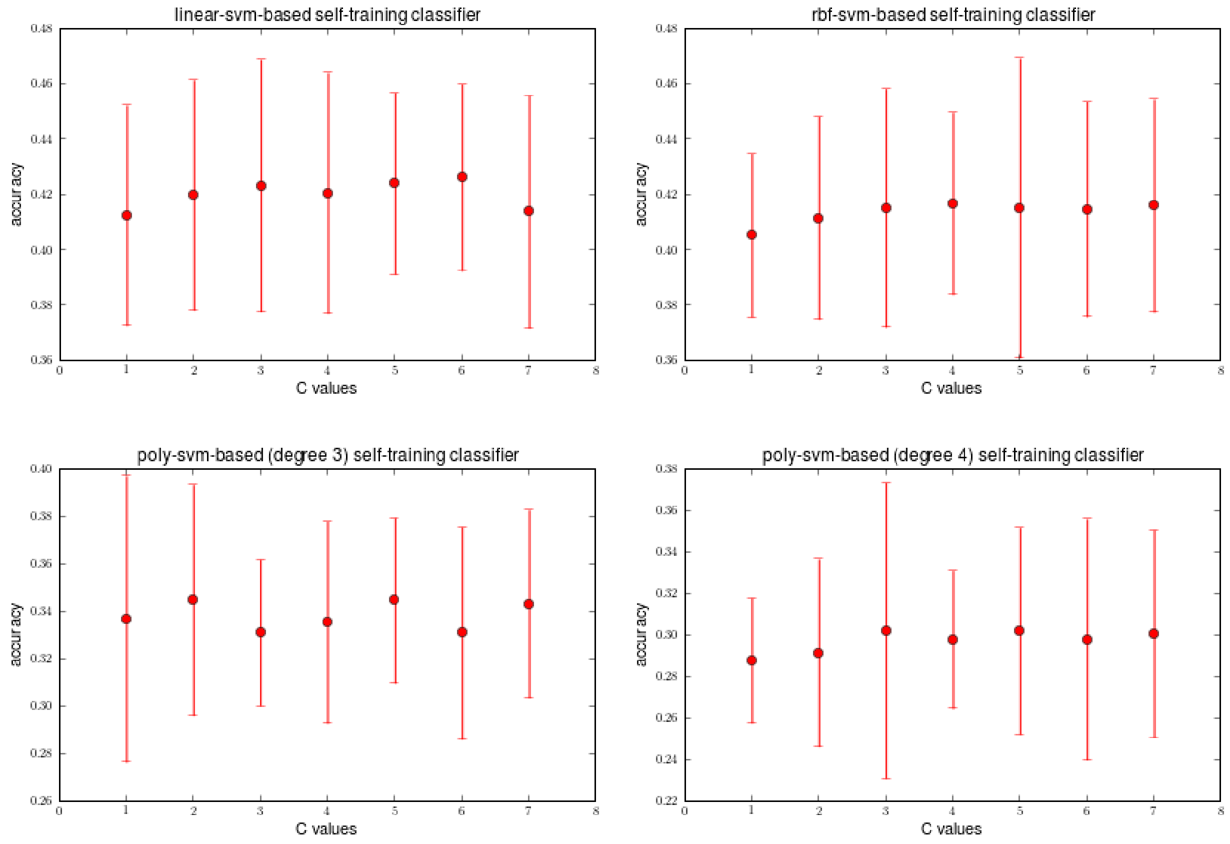

Figure 5: Accuracy of our self-training algorithm using a linear SVM with various C values.

Looking closer at the individual classifiers, we note that the linear- and RBF-kernel SVMs perform much better than the polynomial-kernel SVMs. We also tried the sigmoid kernel, but that performed very poorly compared to all of the above, as it always returned the majority label. Looking again at the linear- and RBF-kernel SVMs, we find that varying C makes a big difference in the overall performance, affecting the classifier by as much as 2–3%. We also varied γ in the case of the RBF-kernel SVM; however, we found no values that performed better than the default γ value ($\gamma = 1/33$, where 33 is the number of features used).

Finally, as a way to visualize these results, we created another time-lapsed video of all sightings between 01/2000 and 08/2012, this time plotting sightings by their classification label, rather than their shape. This video can be viewed at `http://www.cs.umd.edu/~amaloz/ufo/movie2.avi`.

5 Conclusion

In this work, we investigate the feasibility of classifying UFO sightings using basic machine learning techniques. We find that an unclassified approach using K-means does not effectively cluster the sightings; however, semi-supervised techniques appear to be quite successful. We also note that our feature set is rather crude: besides utilizing some basic features provided by all sightings (such as latitude, longitude, shape, and duration), we use a simple bag-of-words approach to construct the feature vectors. Yet, even with such coarse features, we achieved an accuracy of around 43% using self-training with both linear- and RBF-kernel wrapper functions.

We believe this work can be extended in several ways. One such extension would be to improve the features for each sighting. For one, the distance between sightings and both military installations and population centers could be useful features, as there is likely a correlation between said distance and the sighting classification. A more careful feature extraction, instead of a simple "bag-of-words" approach, could also yield more useful features. Lastly, in this work we concentrated on UFO sightings in the contiguous United States; thus, an obvious extension would be to extend this to *all* sightings across Earth.

Code and Data: All the code and data used in this report is available at `http://www.cs.umd.edu/~amaloz/ufo`. In addition, there are two videos. The first video shows UFO sightings between the years 2000 and 2012, categorized by shape. The second shows the same UFO sightings as in the prior video, except this time categorized by the label applied by our best semi-supervised algorithm.

Appendix

A Unsupervised Source Code

```
'''
A collection of unsupervised learning algorithms (currently, a collection of
one).
'''

__AUTHOR__ = 'Alex J. Malozemoff <amaloz@cs.umd.edu>'

import numpy as np

# matplotlib.mlab.entropy is broken, so we just copy over the code and clean
# it up as needed.
def entropy(y):
    n = np.bincount(y.tolist())
    n = n.astype(np.float_)
    n = np.take(n, np.nonzero(n)[0])
    p = np.divide(n, len(y))
    return -1.0 * np.sum(p * np.log(p))

def _v(verbose, string):
    if verbose:
        print string
```

```
def evaluate(labels, clusters):
    '''
    Evaluates our unsupervised algorithm by measuring the entropy of the label
    distribution within each cluster (thanks to Hector for this idea). Returns
    the mean and standard deviation across these entropies.

    Arguments:
        labels - known labels
        clusters - learned clusters
    '''
    clusternames = set(clusters)
    es = np.zeros(len(clusternames))
    for i, cluster in enumerate(clusternames):
        # compute entropy for the classified labels using the given clusters
        es[i] = entropy(labels[clusters == cluster][labels != 0])
    return np.mean(es), np.std(es)

def kmeans(df, K, runs=1, maxiters=None, converge=1e-8, verbose=False):
    '''
    Implementation of the K-means algorithm, as presented in Alg. 34 of CIML.

    Arguments:
        df - DataFrame comprising data to learn
        K - number of clusters
        runs - how many times to repeat the algorithm (to avoid local minima)
        maxiters - number of iterations before we stop
        converge - required difference between two iterations before stopping
    '''
    clusters = []
    objs = []
    _v(verbose, 'Running K-means %d time(s)' % runs)
    for run in xrange(runs):
        z, obj = _kmeans(df, K, maxiters=maxiters, converge=converge,
                         verbose=verbose)
        _v(verbose, 'Run %d: objective = %f' % (run+1, obj))
        clusters.append(z)
        objs.append(obj)
    _v(verbose, 'Objectives: %s' % (repr(objs)))
    _v(verbose, 'Returning run %d' % (np.argmin(objs)+1,))
    return clusters[np.argmin(objs)]

def _kmeans(df, K, maxiters=None, converge=1e-8, verbose=False):
    nexamples = df.shape[0]
    iters = 0
    obj = np.inf

#   # Initialize cluster means to random data points
#   mu = np.array([df.ix[np.random.randint(0, nexamples)] for k in xrange(K)])

    # Use furthest-first heuristic
    _v(verbose, '* running furthest-first heuristic')
    mu = np.empty(K, dtype=object)
    mu[0] = df.ix[np.random.randint(0, nexamples)]
    for k in xrange(1, K):
        ms = [min([np.linalg.norm(df.ix[m] - mu[kp])**2 for kp in xrange(k)]) \
              for m in xrange(nexamples)]
        mu[k] = df.ix[np.argmax(ms)]

    while True:
        iters += 1
        old_obj = obj
        _v(verbose, '* iteration #%d' % iters)
        _v(verbose, '** assigning examples to closest cluster')
        z = np.array([np.argmin([np.linalg.norm(mu_k-df.ix[n]) for mu_k in mu]) \
                      for n in xrange(nexamples)], dtype=int)
        _v(verbose, '** cluster count: %s' % repr(np.bincount(z)))
        _v(verbose, '** re-estimating cluster means')
        mu = np.array([np.mean(df[z==k], axis=0) for k in xrange(K)])
        # compute objective
        obj = 0.0
        for k in xrange(K):
            obj += sum([np.linalg.norm(df.ix[idx] - mu[k])**2 \
                        for idx in df[z == k].index])
        _v(verbose, '** objective: %f' % obj)
        # no more changes detected, so break
        if abs(obj - old_obj) < converge:
            break
        # maxed out number of allowed iterations, so break
        if maxiters is not None and iters >= maxiters:
            break
    return z, obj
```

B Semi-supervised Source Code

```
'''
A collection of semi-supervised learning algorithms (currently, a collection of
one).
'''

__AUTHOR__ = 'Alex J. Malozemoff <amaloz@cs.umd.edu>'

import numpy as np

def _v(verbose, string):
    if verbose:
        print string

def evaluate(trained, y):
    yhat = trained[y.index]
    assert len(yhat) == len(y)
    return sum(i == j for i, j in zip(y, yhat)) / float(len(y))
```

```
def self_training(df, labels, learner, size=None, verbose=False):
    '''
    Implementation of the self-training approach.

    Arguments:
        df - DataFrame composed of data to learn
        labels - column of labels for rows in *df*
        learner - an object with `fit` and `predict_proba` methods
        size - number of items to classify on each iteration
    '''
    nexamples, nfeatures = df.shape
    labels = labels.copy()
    unclsd = labels == 0
    clsd = labels != 0
    n_unclsd = sum(unclsd)
    n_clsd = sum(clsd)
    if size is None:
        size = nexamples
    _v(verbose, '%d classified, %d unclassified' % (n_clsd, n_unclsd))
    while n_unclsd > 0:
        _v(verbose, '* training...')
        learner.fit(df[clsd], labels[clsd])
        test = df[unclsd]
        _v(verbose, '* predicting...')
        pred = learner.predict_proba(test)
        # determine the rows with the best label probability.
        # we store both the index into pred (an array) and the index into test
        # (a dataframe).
        best = [(np.max(row), idx, dfidx) \
                for idx, (row, dfidx) in enumerate(zip(pred, test.index))]
        best.sort(reverse=True)
        bestidx = [idx for _, idx, _ in best[:size]]
        bestdfidx = [dfidx for _, _, dfidx in best[:size]]
        bestlabel = np.array([np.argmax(row) for row in pred[bestidx]]) + 1
        _v(verbose, '* bincount = %s' % np.bincount(bestlabel.tolist()))
        labels.ix[bestdfidx] = bestlabel
        # recalculate the labels still unclassified.
        unclsd = labels == 0
        clsd = labels != 0
        n_unclsd = sum(unclsd)
        n_clsd = sum(clsd)
        _v(verbose, '%d classified, %d unclassified' % (n_clsd, n_unclsd))
    return labels
```

C Evaluation Source Code

```
import numpy as np
import random
import sklearn.svm

import semisupervised as semisup

def _v(verbose, string):
    if verbose:
        print string

def build_train_test_set(labels, size=100):
    labeled = labels[labels != 0]
    nlabeled = len(labeled)
    # extract random labeled elements to use as a test set
    relems = random.sample(labeled.index, size if size < nlabeled else nlabeled)
    test_labels = labeled.ix[relems]
    new_labels = labels.copy()
    # clear out labels in our test set
    new_labels[test_labels.index] = 0
    return new_labels, test_labels

def build_test_set(labels, size=100):
    labeled = labels[labels != 0]
    nlabeled = len(labeled)
    # extract random labeled elements to use as a test set
    relems = random.sample(labeled.index, size if size < nlabeled else nlabeled)
    return labeled.ix[relems]

def cross_validate(df, labels, learner, num=10, size=None, verbose=False):
    trainsets = []
    testsets = []
    nlabels = labels[labels != 0].shape[0]
    setsize = int(np.ceil(nlabels / float(num)))
    tmplabels = labels.copy()
    _v(verbose, "%d-fold cross-validation" % num)
    _v(verbose, "set size = %s" % setsize)
    for _ in xrange(num):
        test = build_test_set(tmplabels, size=setsize)
        testsets.append(test)
        tmplabels = tmplabels.copy()
        tmplabels[test.index] = 0
    for testset in testsets:
        train = labels.copy()
        train[testset.index] = 0
        trainsets.append(train)
    scores = []
    for i in xrange(num):
        _v(verbose, '%d: training...' % i)
        out = semisup.self_training(df, trainsets[i], learner, size=size,
                                    verbose=verbose)
        _v(verbose, '%d: evaluating...' % i)
        score = semisup.evaluate(out, testsets[i])
        _v(verbose, '%d: score = %f' % (i, score))
        scores.append(score)
    return np.mean(scores), np.std(scores)

def test_svm(df, labels, Cs, kernel='linear', num=10, size=None, verbose=False,
             **kwargs):
    means = []
    stdevs = []
    if size is None:
        size = df.shape[0]
    for C in Cs:
        _v(verbose, "C = %f" % C)
        l = sklearn.svm.SVC(probability=True, kernel=kernel, scale_C=True,
                            C=C, **kwargs)
        mean, stdev = cross_validate(df, labels, l, num=num, size=size,
                                     verbose=verbose)
        means.append(mean)
        stdevs.append(stdev)
    return means, stdevs

def test_rbf_svm(df, labels, gammas, C=1.0, num=10, size=None, verbose=False,
                 **kwargs):
    means = []
    stdevs = []
    if size is None:
        size = df.shape[0]
    for gamma in gammas:
        _v(verbose, "gamma = %f" % gamma)
        l = sklearn.svm.SVC(probability=True, kernel='rbf', scale_C=True,
                            C=C, gamma=gamma, **kwargs)
        mean, stdev = cross_validate(df, labels, l, num=num, size=size,
                                     verbose=verbose)
        means.append(mean)
        stdevs.append(stdev)
    return means, stdevs
```