A Combination Of Local Models For Active Learning

Gustavo Arismendi and Anirudh Ravishankar

Department of Computer Science
University of Houston
COSC 7362 Advance Machine Learning
May 3, 2019

Abstract

In this paper we introduce a novel Active Learning measure of uncertainty approach based on multi-dimensional minimum distance. Multiple local models utilize this measure to query an oracle in a classification task. Our experiments show good accuracy results when compared to an existing active learning framework working on the same classification dataset. We discuss the number of initial labeled samples, queries and features and how each of these parameters factored into our final results.

1. Introduction

Active learning is a machine learning technique that seeks to improve classifier efficiency and accuracy by asking queries to an oracle to label a small but informative number of unlabeled samples according to a specific measure of uncertainty (Settles, 2010). The motivation behind active learning include tasks where unlabeled data is abundant and where it's difficult or costly to obtain labels (Dasgupta and Hsu, 2008). Within the major areas of machine learning, active learning shares similarities with both supervised and unsupervised learning. It's similar to supervised learning in that we have some instances with labels used to train a classifier, and like unsupervised learning, we usually have a large amount of unlabeled data that needs to be classified (Dasgupta and Hsu, 2008). Within active learning there are multiple sampling strategies to select the most informative unlabeled sample. Currently most of the algorithms in active learning fall into one of the three following categories: uncertainty sampling, query by committee, and expected reduction (Padmakumar et all, 2018). The focus of our paper is within uncertainty sampling, which uses a selected measure of uncertainty to determine which samples the algorithm will find the most informative to query the oracle for its correct label (Settles, 2010). Some common measurements of uncertainty sampling are least confident, margin, and entropy (Settles, 2010). We developed our own measure of uncertainty based on minimum distance among k-oppositely labelled nearest neighbors and present results of our efforts. The remaining sections of this paper are organized in the following way. In the related works section we discuss previous research and how our approach differs or builds on it. In methodology we give a step-by-step approach to our algorithm. In the experiments section we provide results obtained from using our approach when compared to an existing active learning framework ModAL(Pool-based) (Danka, 2018) . Finally we summarize our results in the conclusion section and present areas where we would like to pursue future work.

2. Related works

A general implementation of an active learning algorithm using uncertainty sampling with a single classifier looks like the algorithm in Figure 1 (Lewis and Gale, 1994).

1. Create an initial classifier
2. While teacher is willing to label examples
    a. Apply the current classifier to each unlabeled example
    b. Find the b examples for which the classifier is least certain of class membership
    c. Have the teacher label the subsample of b examples
    d. Train a new classifier on all labeled examples

Figure 1. An algorithm for uncertainty sampling with a single classifier (Lewis and Gale, 1994).

When compared to this algorithm our approach differs in that we use multiple local models (logistic regression models) resulting from the initial set of labeled samples instead of a single classifier (decision tree). We generate one classifier for each neighborhood of the instance space instead of one global classifier.

In other previous works involving classification (binary and multi-class) tasks researchers have used a wide range of measures of uncertainty. Among the most important and common ones are margin sampling (Scheffer et al., 2001), and entropy (Shannon, 1948). Our algorithm differs from these previous measures in that we use the minimum distance of the k-opposite nearest neighbors as our measure of uncertainty, which we'll explain in more detail in the methodology section.

Finally, when dealing with multiple local models it's common to come up with a way to determine the regions or neighborhoods in which these models will classify new incoming samples. Previous research creates neighborhoods "based on an intuitive idea of locality which uses the Voronoi cell around the prediction point, i.e. all points whose nearest neighbor is the prediction point" (Bennet, 2007). In one way our approach is similar in that we use k-nearest neighbors to come with neighborhoods as well but we differ in that we use an equal number of neighbors for each of the classes present in the dataset. For example, if we are working on a binary classification task we select two neighbors belonging to the positive class and two neighbors belonging to the negative class.

3. Methodology

3.1 Terminology

a. Distance: when mentioning distance in this project we are referring to the Euclidean distance between two points. This distance can be calculated in any number of n dimensions (in the experiment section we compare results with multiple dimensions) . To measure the distance between a point A = (a1, a2,...an) and B = (b1, b2,...bn) with n dimensions we would use the following formula:

$$d(A,B) = d(B,A) = \sqrt{(a_1 - b_1)^2 + (a_2 - b_2)^2 \ldots + (an - bn)^2}$$

b. k - Opposite Nearest neighbors (KONN): group of initially labeled (L) samples that have opposite labels and have closest distance from one another. For example, a KONN group with k = 4 consists of two samples X1 and X2 belonging to one class (0) and two samples X3 and X4 belonging to another class (1). They are grouped together based on the minimum distance between opposing labels, <X1, 0> is closest to <X3, 1>, and <X2, 0> is closest to <X4, 1>. This results in the (X1, X2, X3, X4) KONN.

c. Midpoint(s): the point located in the middle of a KONN, and used to find a specific KONN. For two points A = (a1, a2,...an) and B = (b1, b2,...bn) with n dimensions, the midpoint is calculated by executing:

$$\frac{(a+b)}{2} \text{ for each coordinate } \frac{(a_i+b_i)}{2}, \qquad \text{where i=(1,2,3,...,n)}$$

d. Region of uncertainty: defined as any distance that is less than the minimum distance of any point in a KONN to the midpoint of that KONN. For example, if we have a KONN (X1, X2, X3, X4) where each point has the following distance to the midpoint of that KONN (3,4,2,7) respectively, then our region of uncertainty for that KONN is any unlabeled point (U) that has a distance to the KONN midpoint shorter than 2. In Figure 2, we can see how an unlabeled sample with a distance to the midpoint equal to 1 would fall in our region of uncertainty.

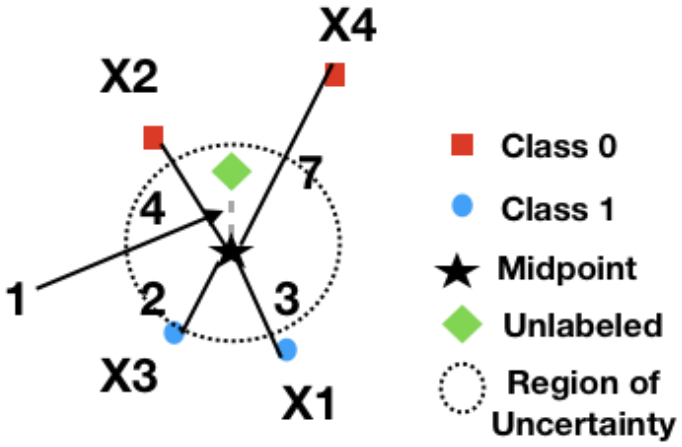

Figure 2. Example of region of uncertainty.

e. Local models: logistic regression models calculated within each KONN.

3.2 Assumptions

During the development of our algorithm we made the following assumptions:

1. Our initial labeled dataset:
    a. Contains a balanced number of instances for each class.
    b. Is a good representation of the underlying distribution for the instance space.
    c. Has numeric features only.
    d. Can be linearly separable.
2. The recursive feature elimination (RFE) method correctly identifies relevant and important features.
3. The decision boundaries for our local models is found between opposite labelled points with closest distance to each other.

3.3 Algorithm

1. ***Select an equal number of positive and negative class samples from the initial labeled samples*.**

    In most active learning algorithms it's common to begin with a small sample of labeled data and a large number of unlabeled data. For our implementation, we used an existing scikit learn dataset and followed different type of splits between labelled and unlabelled data (please see experimentation section for more details). The labeled dataset that we selected for binary classification was the Wisconsin breast cancer dataset (with ~570 samples). We then selected a subset with an equal number of samples of opposite labels (thirty positive and negative samples with their respective classification labels), which constituted around 10% of the entire dataset. After selecting the subset we use the remaining samples as our "unlabeled" samples (i.e 570 - 60 = 510 "unlabeled samples"). Figure 3 shows a small scale example of our initial step.

2. ***Select the desired number of features to use.*** Once we determine the number of features that we wish to work with, we use recursive feature elimination (RFE) from a support vector machine classifier to select the most important features in our subset to represent the data. Through this approach we can select any number of features that we wish (2,5,17…).

3. ***Calculate distance between opposite labelled instances in our labelled subset.*** We select each instance of one class and calculate its distance towards all the instances of the opposite class. All the distances are stored in a distance matrix.

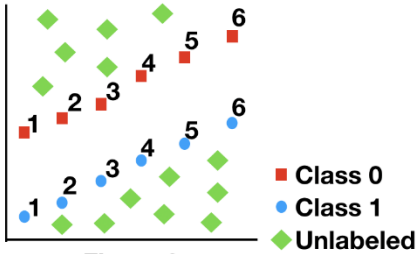

Figure 3

4. ***Select opposite points with the shortest distance between them and calculate their midpoint.*** Each point selects a distinct oppositely labeled point based on minimum distance. If we have ten instances of one label, each will select one instance of the other label based on shortest distance. No one instance can be selected twice. Once we have a pair of oppositely labelled points we calculate their midpoint (Figure 4).

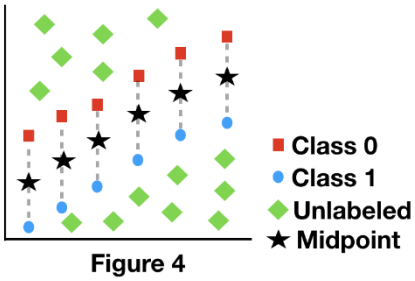

Figure 4

5. ***For every midpoint point choose KONN instance points.*** Select the closest k-opposite nearest neighbor instances around the midpoint, not including the two opposite instances used to calculate the midpoint in Figure 5. We do this to increase our coverage of the instance space.

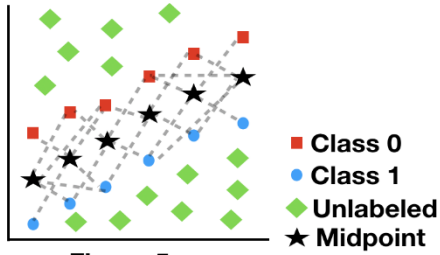

Figure 5

6. ***Generate local models for classification.*** Store these models using the midpoint as a key for locating a specific model. Keep generating more local models until we exhaust all the midpoints. In Figure 6 we see how each midpoint selects four points (two of each class).

7. ***Select uncertain samples and query oracle.*** Once we have the local models and the number of queries, we select a sample that falls into the area of uncertainty for a local model and ask the oracle for its correct label. We add the newly labeled sample to our L dataset while simultaneously removing it from our U dataset. We proceed to retrain the local model where the newly labeled sample was added and repeat this process until we run out of queries (Figure 7).

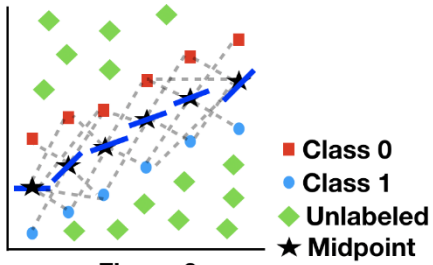

Figure 6

8. ***Feed remaining unlabeled samples to the local models until all instances are labeled.***

When an unlabeled instance point comes, we calculate its distance to all the available midpoints. We proceed to locate the closest midpoint to it and use the local model for that midpoint to assign a label to the instance. We repeat this process until we are done classifying all unlabeled instances (Figure 7).

Implementation:
Q → Number of queries to oracle
L → Labeled data
U → Unlabeled data
Train logistic regression classifiers on L → models M

Repeat (Q)
  Select an uncertain sample x from U according to closest local M.
  Query the sample and obtain label y.
  Add (x, y) to the set of labeled examples for the right local model.
  Remove x from U.
  Retrain specific local
End

Classify remaining U data with M.

Figure 7

3.4 Advantages

1. The algorithm uses an initial random sampling of the labeled dataset. This can avoid falling into the sampling-bias problem that occurs in most conventional active learning algorithm (Dasgupta and Hsu, 2008). However, we recognize that random sampling could also be a disadvantage (please see disadvantage 3).
2. The use of minimum distance between oppositely labeled points to determine midpoints and create the region of uncertainty within KONNs is straightforward, flexible, and easy to implement.
3. The algorithm could be implemented using different types of local models to work with multi-class classification problems and deal with data that is not linearly separable.

3.3 Disadvantages

1. It may be difficult or not possible to obtain a starting labeled dataset with an equal number of opposite class samples (or with the opportunity to create this). Therefore, assumption 1a can be difficult to accomplish in some datasets.
2. Due to the random initial sampling of the labeled instances, there is a possibility that all or most of our labelled samples belong to a small region of the instance space and that other regions are poorly represented. This will most likely result in poor accuracy.
3. We need to be cautious in selecting the number of local models to generate. In one scenario we could generate too many local models that result in overfitting. In another scenario the number of local models could insufficient to correctly label the instance, which could lead to underfitting.

4. Experiments

In this section we'll explain how we used the breast cancer dataset to run experiments for the implementation of our algorithm and how our results compared to results obtained from using the same dataset on the modAL active learning framework. Our main focus will be in how we varied the following parameters during experimentation:

- Initial size of labeled samples (L).
- Number of k-opposite nearest neighbors (k).
- Number of features.
- Number of queries to oracle (Q).

1. Initial size of labeled samples (L): the breast cancer dataset has a total of 569 instances. From this total we selected 20, 60, and 100 total labeled samples with a split of 50% of one class and 50% of another class.

2. Number of k-opposite nearest neighbors (k): for this parameter we kept the k=4 for all of our experiments.

3. Number of features: our features parameters were tested using 2, 5, 7, 10 and even 20 features for a dataset that has a total of 30 features to choose from. All of these features were selected used recursive feature elimination (RFE).

4. Number of queries to oracle (Q): The number of queries that our algorithm asked the oracle varied between 0, 5, 10, 15, and 20. We stopped at a twenty query maximum to respect the general active learning assumption that labeled samples are costly or difficult to obtain.

sklearn's Breast cancer, Instanes: 569**, Features:** 30**, y =** [0,1**]**

- **Number of initial Labeled instances picked**: 20 (10 from each class)

	Number of Features				
Number of queries	**2**	**5**	**7**	**10**	**20**
0	**0.857923**	**0.899817**	**0.899817**	**0.914389**	**0.888888**
5	**0.852941**	**0.875**	**0.90625**	**0.880514**	**0.852941**
10	**0.857142**	**0.892393**	**0.903525**	**0.899814**	**0.855822**
15	**0.855805**	**0.926966**	**0.921348**	**0.889513**	**0.855822**
20	**0.865784**	**0.924385**	**0.901701**	**0.887218**	**0.855822**

- **Number of initial Labeled instances picked**: 60 (30 from each class)

Number of queries	Number of Features				
	2	5	7	10	20
0	0.860510	0.935166	0.909626	0.929273	0.893909
5	0.791666	0.880952	0.801587	0.916666	0.886904
10	0.847695	0.927855	0.809619	0.919839	0.891783
15	0.876518	0.929149	0.807692	0.917004	0.892712
20	0.885480	0.930470	0.809815	0.899795	0.887983

- **Number of initial Labeled instances picked**: 100 (50 from each class)

Number of queries	Number of Features				
	2	5	7	10	20
0	0.835820	0.878464	0.810234	0.899786	0.933901
5	0.816810	0.872844	0.808189	0.900862	0.9375
10	0.823529	0.875816	0.806100	0.906318	0.941176
15	0.828193	0.894273	0.806167	0.920704	0.936263
20	0.826280	0.895322	0.728285	0.913140	0.936263

Conventional Active Learning(ModAL):

Number of queries	Accuracy
0	0.627
5	0.8875
10	0.7663
15	0.8787
20	0.8858

5. Figures / Statistics / Conclusions (insights from results)

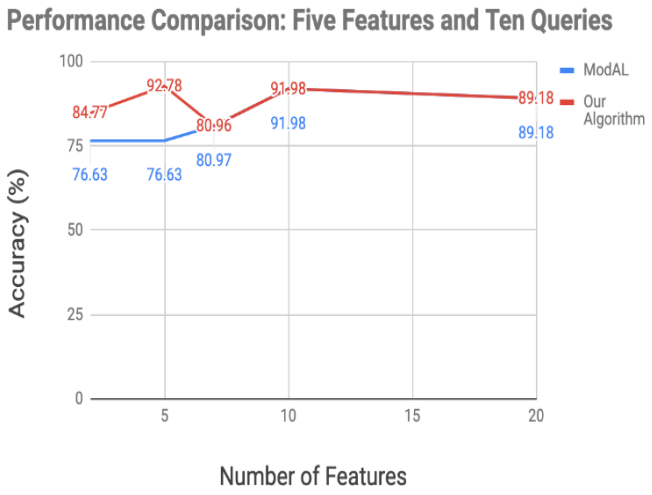

Figure 8

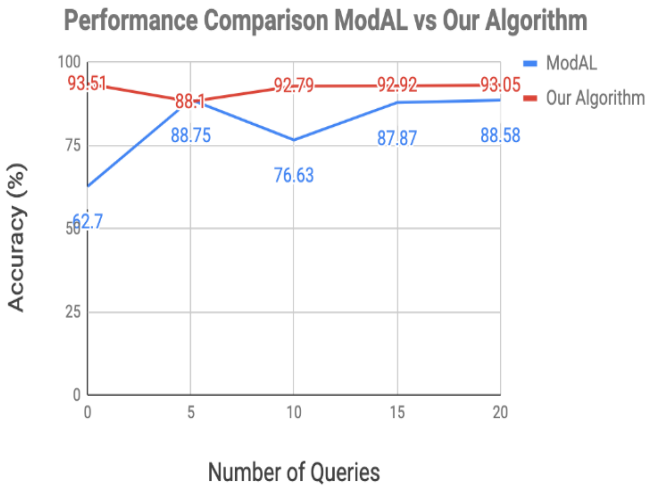

Figure 9

- In Figures 8 and 9, we can see that our model performed well when compared to the existing Pool-based Active Learning (ModAL) project. We believe the main reasons to be (1) the use of several local models instead of single global model, and (2) the retraining of the local models after queries.

- Looking at Figure 8 we can see that our accuracy is higher than ModAL when we have a low number of features, and as we increase the features we see that both algorithms work very similarly.

- In Figure 9 we see that our algorithm performs well even with no queries, and as we begin to query we see a decline in accuracy at first but then a steady accuracy regardless of the number of features selected. We believe that in some cases the local models may be over-fitting the training data and as we increase the number of queries we see an initial accuracy drop that proceeds to go back to normal and flatten out as we increase the number of queries.

Next, we'll give an explanation of the accuracy of our algorithm depending on the initial sample size (or number of local models).

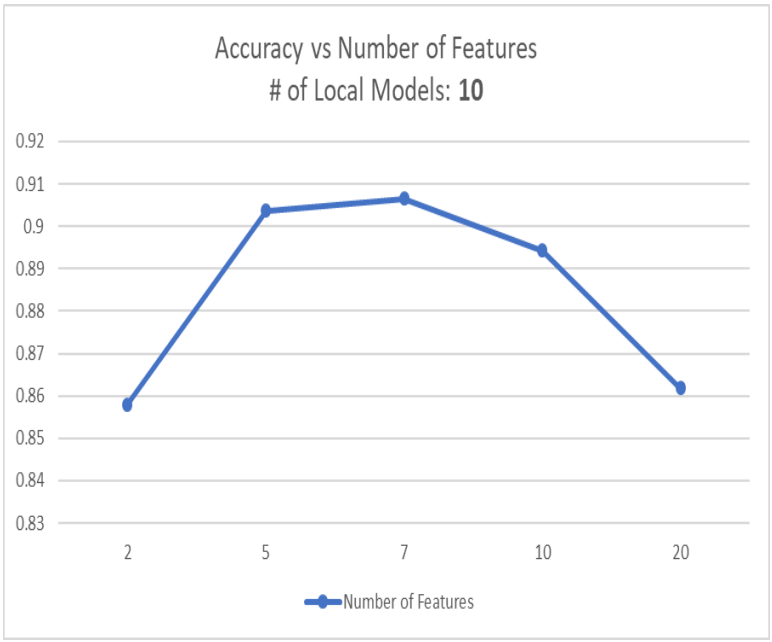

Figure 10

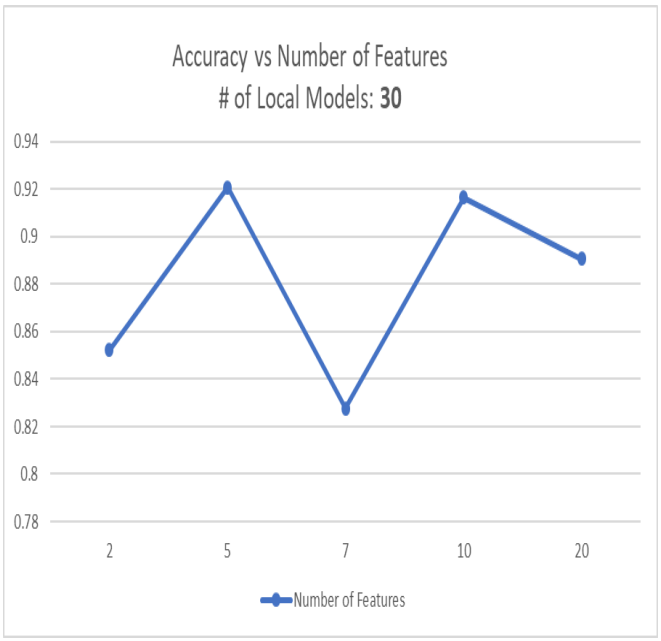

Figure 11

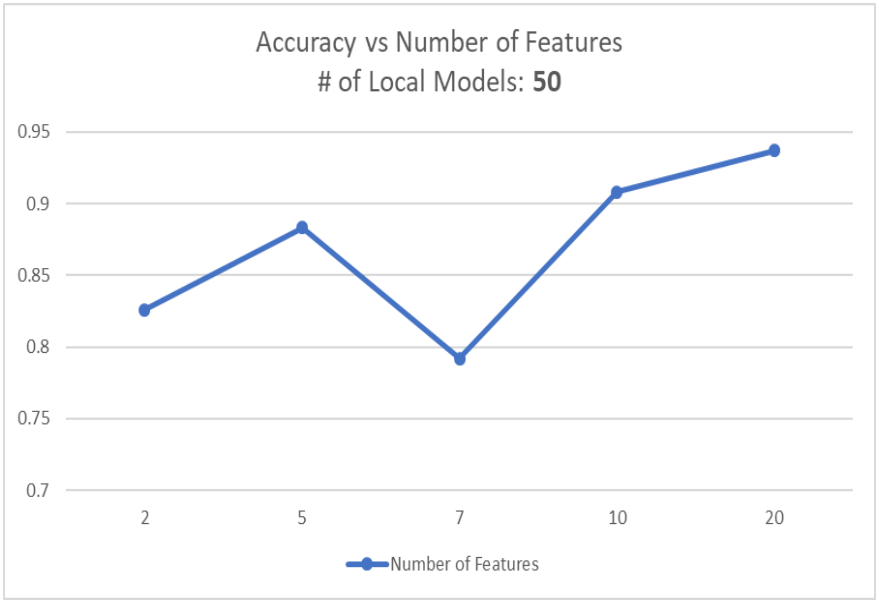

Figure 12

- In Figures 10, 11, and 12 above, we can see for the most part that the classification seems to be good when we choose the number of features to be 5 regardless of the initial size of the labeled dataset (20, 60, 100).
- In general, it seems that the small (Figure 10) and medium size (Figure 11) number of local models were more accurate with a lower number of features, while the larger number of local models (Figure 12) benefited from having a larger number of features.
- Our algorithm is implemented in a way that various feature selection methods can be used and we leave it to each researcher to choose the method that will give them number of features that they seem best.

Accuracy of our algorithm in comparison with the number of local Models

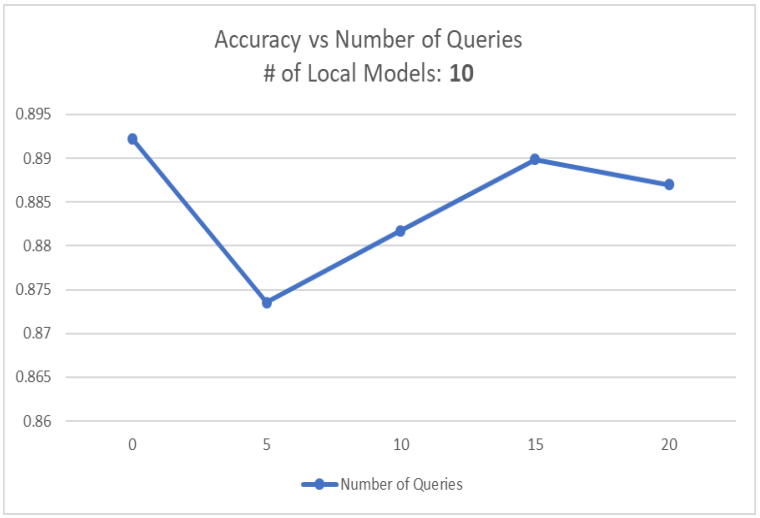

Figure 13

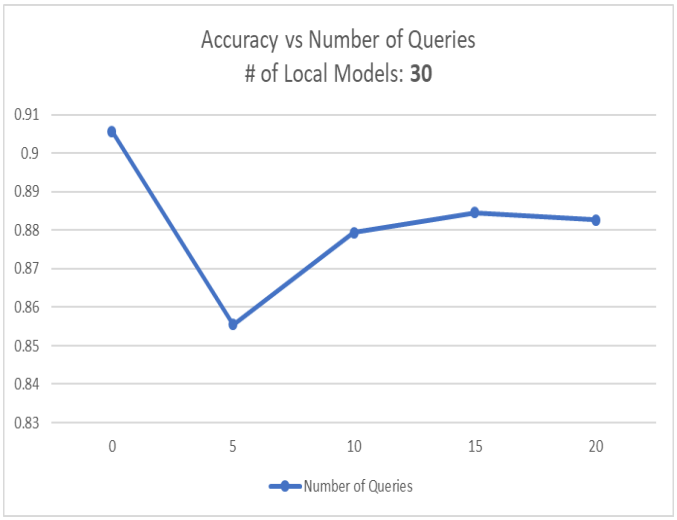

Figure 14

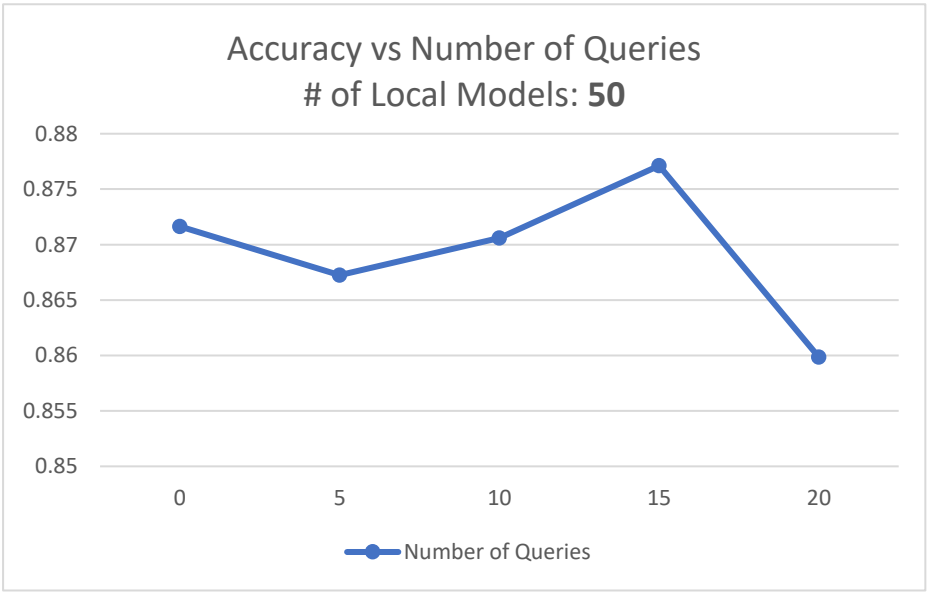

Figure 15

- Figures 13, 14, and 15 help us see the best size of the initial labeled dataset in relation to the entire dataset.
- Our algorithm uses randomness to pick the initial labeled dataset for training the local models, hence there is a possibility that the data could belong to a nearby neighborhood and ignores the other neighborhoods in the instance space.
- The accuracy seems to do well as we increase the number of queries in Figures 13 and 14, however, Figure 15 seems to show us that if we have too many local models the higher number of queries will result in overfitting the data.
- Figure 15 shows us that once we select more than 10% of the whole dataset for our initial labeled sample set, the number of local models that we create will result in overfitting.

5. Further Work

For future work we'd like to concentrate on the following areas: multi-class classification problems, non-linearly separable datasets, other type of classifiers for local models, the relation between number of queries for specific dataset sizes, the relation between the number of queries and number of classes present in dataset, a more sophisticated measure of uncertainty, and experimenting with different strategies to select the initial set of labeled samples from which to start working with.

For our first point, we'd like to work with classification tasks that have more than two classes to see how our algorithm performs. We currently worked with two different binary classification datasets to make sure that our algorithm was working correctly. Another area that we'd would like to experiment with is with data were the decision boundary is not linearly separable to see how our algorithm would perform in those situations.

During this project we concentrated on using logistic regression for our local models. In the future we'd like to use an assortment of classifiers (support vector machines is one of our top priorities) to compare performance. We are also interested in studying the relationship between number of queries depending on the initial dataset size and on the number of class present on the dataset to have a more efficient process when selecting the what number of queries results in best performance for a specific dataset.

Finally, we'd like to implement other more measures of uncertainty that may be a combination of our neighborhoods with the Voronoi cell neighborhoods done by Bennet (Bennet, 2007) or that take advantage of the clustering framework developed by Dasgupta and Hsu (Dasgupta and Hsu, 2008) .

References

- Active Learning: Literature Survey, by Burr Settles, Technical Report, University of Wisconsin Madison, 2010.
- Hierarchical Sampling with Active Learning by Dasgupta and Hsu, Proceedings of the 25th International Conference on Machine Learning, Helsinki, Finland, 2008.
- Aishwarya Padmakumar, Peter Stone, Raymond J. Mooney In Proceedings of the 2018. Learning a policy for Opportunistic Active Learning. Conference on Empirical Methods in Natural Language Processing (EMNLP-18), Brussels, Belgium, November 2018.
- Meng Fang, Yuan Li, and Trevor Cohn. 2017. Learning how to active learn: A deep reinforcement learning approach. In Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing. ACL.
- D. Lewis and W. Gale. A sequential algorithm for training text classifiers. In Proceedings of the ACM SIGIR Conference on Research and Development in Information Retrieval, pages 3–12. ACM/Springer, 1994.
- T. Scheffer, C. Decomain, and S. Wrobel. Active hidden Markov models for information extraction. In Proceedings of the International Conference on Advances in Intelligent Data Analysis (CAIDA), pages 309–318. Springer-Verlag, 2001.
- C.E. Shannon. A mathematical theory of communication. Bell System Technical Journal, 27:379–423,623–656, 1948.
- Paul N. Bennett. Neighborhood-Based Local Sensitivity. *Proceedings of ECML 2007* | September 2007. Published by Springer Verlag.
- Tivadar Danka. 2018. ModAL: A modular active learning framework for Python3. https://modal-python.readthedocs.io/en/latest/index.html

Appendix: Algorithm Source Code

```
from sklearn import datasets
from sklearn.feature_selection import RFE
from sklearn.svm import LinearSVC
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
import numpy as np
import pandas as pd
from math import sqrt
from sklearn.preprocessing import StandardScaler

# Loading the dataset
cancer_data = datasets.load_breast_cancer()
X_raw = cancer_data['data']
y_raw = cancer_data['target']
complete_data = np.insert(X_raw, 30, y_raw, axis=1)

# Standardize data
sc = StandardScaler()
sc.fit(X_raw)
X_raw = sc.transform(X_raw)

# Use RFE for feature selection
svm = LinearSVC(random_state=0, C=1, max_iter=100000)
rfe = RFE(svm, 20)
fit_cancer_data = rfe.fit(X_raw, y_raw)
transform_cancer_data = rfe.transform(X_raw)

# creating a dataframe out of numpy ndarrray
X_df = pd.DataFrame(X_raw)
X_df['y'] = cancer_data['target']
y_df = pd.DataFrame(y_raw)

# Split into training and test data
X_train, X_test, y_train, y_test = train_test_split(X_df, y_df, test_size=0.3, random_state=0)

# Getting the best features
best_features = list()
a = 0
for i in rfe.ranking_:
    if i == 1:
        best_features.append(a)
    a += 1
best_features.append('y')

# Get list of all features
num_orig_feat = X_raw.shape[1]
orig_feat_index = list()
for i in range(num_orig_feat):
    orig_feat_index.append(i)

# Drop extra features ( original - best_features)
drop_features = list(set(orig_feat_index) - set(best_features))
X_train_reduced_feat = X_train.drop(drop_features, axis=1)
X_test_reduced_feat = X_test.drop(drop_features, axis=1)
```

```
# Get 60 samples with labels (30 of one class and 30 of other) from the training set
all_zero_labels = X_train_reduced_feat[X_train_reduced_feat['y'] == 0]
all_one_labels = X_train_reduced_feat[X_train_reduced_feat['y'] == 1]
subset_labeled_zero = all_zero_labels.drop(['y'], axis=1).head(50)
subset_labeled_one = all_one_labels.drop(['y'], axis=1).head(50)
subset_train_labeled_samples = pd.concat([subset_labeled_zero, subset_labeled_one])

# Get remaining samples without labels
remaining_labeled_zero = all_zero_labels.tail(all_zero_labels.shape[0] - subset_labeled_zero.shape[0])
remaining_labeled_one = all_one_labels.tail(all_one_labels.shape[0] - subset_labeled_one.shape[0])
remaining_train_labeled_samples = pd.concat([remaining_labeled_zero, remaining_labeled_one])
final_unlabeled_train_data = remaining_train_labeled_samples.drop(['y'], axis=1)
X_test_unlabeled = X_test_reduced_feat.drop(['y'], axis=1)

#creating list of tuples of the feature values in both classes 0 and 1
tuples_zero = [tuple(x) for x in subset_labeled_zero.values]
tuples_one = [tuple(x) for x in subset_labeled_one.values]

#renaming the columns of the dataframe in sequence[ 0,1,2....length of selected features]
new_column_name_training = list()
for num in range(len(best_features) - 1):
    new_column_name_training.append(num)
final_unlabeled_train_data.columns = new_column_name_training

new_column_name_X_test = list()
for num in range(len(best_features) - 1):
    new_column_name_X_test.append(num)
X_test_unlabeled.columns = new_column_name_X_test

# creating the final unlabeled data from the one's that have not been picked for training samples along with the test data
final_unlabeled_train_data_tuples = [tuple(x) for x in final_unlabeled_train_data.values]
X_test_unlabeled_coords_tuples = [tuple(x) for x in X_test_unlabeled.values]
final_unlabeled_train_data['coords'] = final_unlabeled_train_data_tuples
X_test_unlabeled['coords'] = X_test_unlabeled_coords_tuples

# Get distance of opposite labels
subset_labeled_zero['Coord'] = tuples_zero
subset_labeled_one['Coord'] = tuples_one
zero_index_values = list(subset_labeled_zero.index.values)
one_index_values = list(subset_labeled_one.index.values)

# We need to calculate the distance between the opposite labels and store it.
row_values = pd.Index(zero_index_values, name='rows')
column_values = pd.Index(one_index_values, name='columns')
dist_matrix = pd.DataFrame(data=0, index=row_values, columns=column_values)

#picking the closest - oppositely labeld instances
for row, column in subset_labeled_zero.iterrows():
    for row2, column2 in subset_labeled_one.iterrows():
        sum_squares = 0
        for i in range(len(best_features) - 1):
            sum_squares += (column['Coord'][i] - column2['Coord'][i]) ** 2
        dist_matrix.loc[row, row2] = sqrt(sum_squares)
```

```
# Get min distance value for each row. Problem: may not pick best min global value for each row. i.e: row 1 may pick
# column 4 as its min value but it turns out that column 4 and row 3 were closer but row 1 got to pick first.
# NOTE: mean doesn't work well for k-neighbors, using midpoint instead.

min_dist_list = list()
for row3, column3 in dist_matrix.iterrows():
    v = dist_matrix.values
    global_min_dist_index = np.nanargmin(v)
    i, j = [x[0] for x in np.unravel_index([global_min_dist_index], v.shape)]
    zero_one_index_pair = (dist_matrix.index[i], dist_matrix.columns[j])
    min_dist_list.append(zero_one_index_pair)
    dist_matrix.loc[zero_one_index_pair[0], zero_one_index_pair[1]] = np.nan

# Find midpoint to choose k-nearest neighbor.
midpoints = list()
get_features = list(set(best_features) - set(list(['y'])))
for dist_pair in min_dist_list:
    zero_label_coord = subset_labeled_zero.loc[dist_pair[0]]
    one_label_coord = subset_labeled_one.loc[dist_pair[1]]
    mp = list()
    for i in get_features:
        mp.append((zero_label_coord[i] + one_label_coord[i]) / 2)
    midpoints.append(tuple(mp))

#renaming the columns for zero-labeld subset as well as one-labeled subset
renamed_subset_zero = subset_labeled_zero
new_column_name_zero = list()
for num in range(len(best_features) - 1):
    new_column_name_zero.append(num)
new_column_name_zero.append('Coord')
renamed_subset_zero.columns = new_column_name_zero

renamed_subset_one = subset_labeled_one
new_column_name_one = list()
for num in range(len(best_features) - 1):
    new_column_name_one.append(num)
new_column_name_one.append('Coord')
renamed_subset_one.columns = new_column_name_one

# Find 2 nearest zero neighbors to each midpoint
k_nearest_zeros = dict()
for point_zero in midpoints:
    for (a, b) in min_dist_list:
        k_zero_points_dist = list()
        for index, row in subset_labeled_zero.iterrows():
            if index != a:
                v = 0
                for i in range(len(best_features) - 1):
                    v += ((row[i] - point_zero[i]) ** 2)
                k_zero_points_dist.append((sqrt(v), index))
        sorted_list = sorted(k_zero_points_dist, key=lambda x: x[0])
        k_nearest_zeros[point_zero] = sorted_list[:2]
```

```
# Find 2 nearest one neighbors to each midpoint
k_nearest_ones = dict()
for point_one in midpoints:
    for (a, b) in min_dist_list:
        k_one_points_dist = list()
        for index, row in subset_labeled_one.iterrows():
            if index != a:
                u = 0
                for j in range(len(best_features) - 1):
                    u += ((row[j] - point_one[j]) ** 2)
                k_one_points_dist.append((sqrt(u), index))
        sorted_list = sorted(k_one_points_dist, key=lambda x: x[0])
        k_nearest_ones[point_one] = sorted_list[:2]


# Combine zeros and one labels
combine_nearest_neigh = [k_nearest_zeros, k_nearest_ones]
nearest_neigh = dict()
for k in k_nearest_zeros.keys():
    nearest_neigh[k] = tuple(nearest_neigh[k] for nearest_neigh in combine_nearest_neigh)

# Create logistic regression model
local_classifiers = dict()
for points in midpoints:
    zero_neighbors = list()
    one_neighbors = list()
    zero_values_midpoint = k_nearest_zeros.get(points)
    one_values_midpoint = k_nearest_ones.get(points)
    zero_neighbors.append([b for (a, b) in zero_values_midpoint])
    one_neighbors.append([b for (a, b) in one_values_midpoint])
    all_neighbors_zero = subset_labeled_zero.loc[zero_neighbors[0], :]
    all_neighbors_one = subset_labeled_one.loc[one_neighbors[0], :]
    all_neighbors = pd.concat([all_neighbors_zero, all_neighbors_one])
    X_neighbors_data = all_neighbors.to_numpy()[:4, :(len(best_features) - 1)]
    y_neighbors_labels = list()
    for index, row in all_neighbors.iterrows():
        y_neighbors_labels.append(X_train.loc[index]['y'])
    y_neighbors_data = np.asarray(y_neighbors_labels)
    #creating the logistic regressor local model for each Midpoint
    classifier = LogisticRegression(C=1, random_state=0, solver='lbfgs', max_iter=1000)
    classifier.fit(X_neighbors_data, y_neighbors_data)
    local_classifiers[points] = classifier


# For each data point in train set calculate the distance to each midpoint
# TODO: Rename this variable to complete_train_data??
complete_test_data = pd.concat([final_unlabeled_train_data, X_test_unlabeled])
test_midpoint_least_dist_list = list()
```

```
#store the nearest Midpoint for every test data
for index, row in complete_test_data.iterrows():
    test_midpoint_list = list()
    for point in midpoints:
        dist = 0
        for i in range(len(best_features) - 1):
            dist += ((row[i] - point[i]) ** 2)
        test_midpoint_list.append((sqrt(dist), row['coords'], index, point))
    sorted_list = sorted(test_midpoint_list, key=lambda x: x[0])
    test_midpoint_least_dist_list.append(sorted_list[0][0:4])


# Iterating over the list(test_midpoint_least_dist_list) of info as previously mentioned
# matching the right local model by equating the values of mid points in the inner for-loop
# storing the result of the prediction of all the test points in a list(y_pred)
y_pred = list()
num_queries = 0
query_constraint = 20
final_remaining_unlabeled = test_midpoint_least_dist_list.copy()

for (dist, test_points, ind, mid_points) in test_midpoint_least_dist_list:
    closest_points = nearest_neigh[mid_points]
    zero_distances = [a[0] for a in closest_points[0]]
    one_distances = [b[0] for b in closest_points[1]]
    dist_clos_pts = zero_distances + one_distances
    min_dist_val = min(dist_clos_pts)
    if dist < min_dist_val and num_queries < query_constraint:
        # Ask annotator
        y_label_test_point = X_df.loc[ind]['y']
        element_to_remove = 0
        for item in test_midpoint_least_dist_list:
            if item[2] == ind:
                element_to_remove = item
        # index_elem_to_rem = test_midpoint_least_dist_list.index(element_to_remove)
        final_remaining_unlabeled.remove(element_to_remove)
        X_test_pnt_label = X_df.loc[ind][best_features[:len(best_features)-1]]
        X_test_coord = tuple(x[1] for x in X_test_pnt_label.iteritems())
        rename_indexes = [i for i in range(len(best_features) - 1)]
        X_test_pnt_label.index = rename_indexes
        X_test_pnt_label['Coord'] = X_test_coord
        num_queries += 1

        current_local_model = local_classifiers[mid_points]
        if y_label_test_point == 1 or y_label_test_point == 1.0:
            k_nearest_ones[mid_points].append((dist, ind))
            subset_labeled_one = subset_labeled_one.append(X_test_pnt_label)
        else:
            k_nearest_zeros[mid_points].append((dist, ind))
            subset_labeled_zero = subset_labeled_zero.append(X_test_pnt_label)
```

```
        zero_neighbors = list()
        one_neighbors = list()
        zero_values_midpoint = k_nearest_zeros.get(mid_points)
        one_values_midpoint = k_nearest_ones.get(mid_points)
        zero_neighbors.append([b for (a, b) in zero_values_midpoint])
        one_neighbors.append([b for (a, b) in one_values_midpoint])

        all_neighbors_zero = subset_labeled_zero.loc[zero_neighbors[0], :]
        all_neighbors_one = subset_labeled_one.loc[one_neighbors[0], :]
        all_neighbors = pd.concat([all_neighbors_zero, all_neighbors_one])
        X_neighbors_data = all_neighbors.to_numpy()[:all_neighbors.shape[0], :(len(best_features) - 1)]
        y_neighbors_labels = list()
        for index, row in all_neighbors.iterrows():
            y_neighbors_labels.append(X_df.loc[index]['y'])
        y_neighbors_data = np.asarray(y_neighbors_labels)

        current_local_model.fit(X_neighbors_data, y_neighbors_data)
        local_classifiers[mid_points] = current_local_model

# for other test data that is certain or the number of queries has reached its limit, we use the local classifier to get the label
for (dist, test_points, ind, mid_points) in final_remaining_unlabeled:
    for key in local_classifiers:
        ctr = 0
        for i in range(len(key)):
            if mid_points[i] == key[i]:
                ctr += 1
        if ctr == len(key):
            m = local_classifiers[key]
            vect = np.asarray([test_points])
            pred = m.predict(vect)
            y_pred.append((pred, ind))
            break


unlabeled_y_labels = remaining_train_labeled_samples['y']
final_unlabeld_y_labels = pd.concat([unlabeled_y_labels, y_test])

# Preparing test data and predicted data for accuracy
y_final_pred = list()  # List of predicted labels
for (i, j) in y_pred:
    y_final_pred.append((int(i[0]), j))

y_final_test = list()  # List of true labels
for (dist, test_points, ind, mid_points) in final_remaining_unlabeled:
    y_label = X_df.loc[ind]['y']
    y_final_test.append((int(y_label), ind))
```

```
ctr = 0
for f, b in zip(y_final_test, y_final_pred):
    if f[1] == b[1] and f[0] == b[0]:
        ctr += 1

# Printing the accuracy
acc = float(ctr / len(y_final_test))
print("ACC: ", acc)
print("CTR: ", ctr)
print("LEN: ", len(y_final_test))
```