Assignment 1: Supervised Learning

Haoran Yang – ID: hyang412 – Email: hyang412@gatech.edu

The algorithms used in this project:

In this project, we will explore following ML algorithms by two classification tasks.

1. Decision trees (DecisionTreeClassifier from sklearn)
2. Neural Network (MLPClassifier from sklearn)
3. AdaBoost Tree (AdaBoostClassifier from sklearn)
4. Support Vector Machines (svm.SVC from sklearn)
5. KNN (KNeighborsClassifier from sklearn)

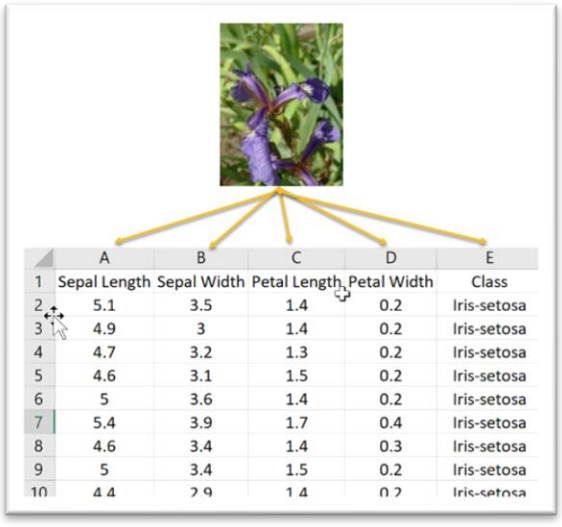
All of the mentioned model will be implemented by python package sklearn.

Scikit-learn (sklearn) is the most useful and robust library for machine learning in Python. It provides a selection of efficient tools for machine learning and statistical modeling including classification, regression, clustering and dimensionality reduction via a consistence interface in Python

Two datasets and tasks in this project:

- 3-target Classification task by Iris data
- 10-target Classification task by MNIST handwriting data

The **Iris data** is perhaps the best-known database to be found in the pattern recognition literature. The data set consists of 50 samples from each of three species of Iris (Iris setosa, Iris virginica and Iris versicolor). Four features were measured from each sample: the length and the width of the sepals and petals, in centimeters. Our task will be to predict the right species by these 4 features.



	A	B	C	D	E
1	Sepal Length	Sepal Width	Petal Length	Petal Width	Class
2	5.1	3.5	1.4	0.2	Iris-setosa
3	4.9	3	1.4	0.2	Iris-setosa
4	4.7	3.2	1.3	0.2	Iris-setosa
5	4.6	3.1	1.5	0.2	Iris-setosa
6	5	3.6	1.4	0.2	Iris-setosa
7	5.4	3.9	1.7	0.4	Iris-setosa
8	4.6	3.4	1.4	0.3	Iris-setosa
9	5	3.4	1.5	0.2	Iris-setosa
10	4.4	2.9	1.4	0.2	Iris-setosa

Figure 1 - Iris Data

The **MNIST data** is a large database of handwritten digits that is commonly used for training various image processing systems, is also widely used for training and testing in the field of machine learning. It was created by "re-mixing" the samples from NIST's original datasets. The black and white images from NIST were normalized to fit into a 28x28 pixel bounding box and anti-aliased, which introduced grayscale levels. The MNIST database contains 70,000 images.

Figure 2 - MNIST handwriting data

Details for 5 supervised learning algorithms used in this project

Decision trees

1. Algorithm Details

Decision Trees are a class of very powerful Machine Learning model cable of achieving high accuracy in many tasks while being highly interpretable. What makes decision trees special in the realm of ML models is really their clarity of information representation. The “knowledge” learned by a decision tree through training is directly formulated into a hierarchical structure. This structure holds and displays the knowledge in such a way that it can easily be understood, even by non-experts.

Some common parameters we can adjust when training the model.

- criterion: {“gini”, “entropy”}: The function to measure the quality of a split. Supported criteria are “gini” for the Gini impurity and “entropy” for the information gain.
- max_depth: The maximum depth of the tree.
- min_samples_split: The minimum number of samples required to split an internal node
- min_samples_leaf: The minimum number of samples required to be at a leaf node. A split point at any depth will only be considered if it leaves at least min_samples_leaf training samples in each of the left and right branches. This may have the effect of smoothing the model, especially in regression.

For the first parameter, it depends on how we define “purity”. Decision trees recursively split features by their target variable’s “purity”. The algorithm is designed to find the optimal point of the most predictive feature to split one dataset into two. In a general sense “purity” can be

thought of as how homogenized a group is. But homogeneity can mean different things depending on which mathematical backbone your decision tree runs on. The 2 most popular backbones for decision tree's decisions are Gini Index and Information Entropy.

For the rest of the parameters, they are different forms of pruning, which to avoid overfitting. We will mainly adjust the max_depth in our exploration.

2. Implementation in task 1

We split the 150-instance dataset into training set (112 - 75%) and testing set (38 - 25%). Since this dataset is not big, intuitively overfitting will not be a severe problem here.

Fixed parameters: { criterion = 'gini', max_depth = 10, min_samples_split = 2, min_samples_leaf = 1 }

We record the learning curve:

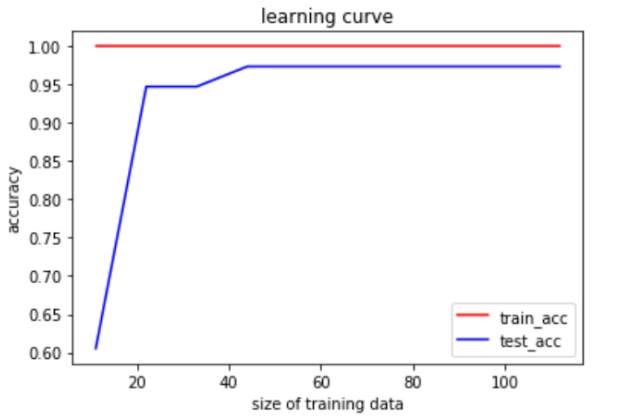


Figure 3 - learning curve

As we can observe, the training accuracy is always 100% which means the decision tree can perfectly predict the training data regardless of the size of training data. On the other hand, if the size of training data is smaller than 20, the test accuracy is much lower. And when the size of training data is bigger than 40, the test accuracy keeps as ~97%.
Then we fixed the size of training data as maximum 112, check how the max_depth controls the overfitting by pruning.

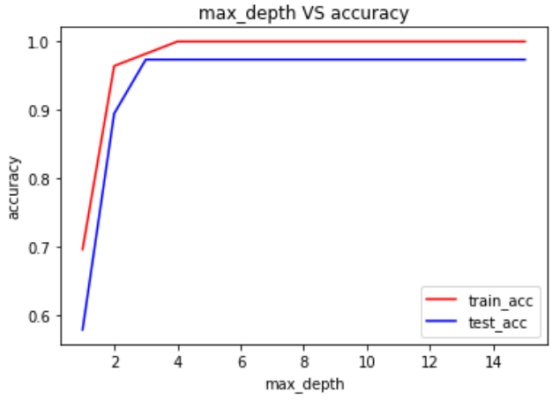


Figure 4 - max_depth VS accuracy

Since the train and test accuracy keep as 1 and 0.97 with 4+ max_depth, we do not observe an obvious overfitting.

In summary, the Iris data is not big in both data size and feature size, which enable us to represent the mapping function in a relevant simple way. Also, it is naturally unlikely to overfit.

3. Implementation in task 2

We split the 70000-instance dataset into training set (52500 - 75%) and testing set (17500 - 25%).

Fixed parameters: { criterion = 'gini', max_depth =18, min_samples_split = 2, min_samples_leaf = 1 }

We record the learning curve:

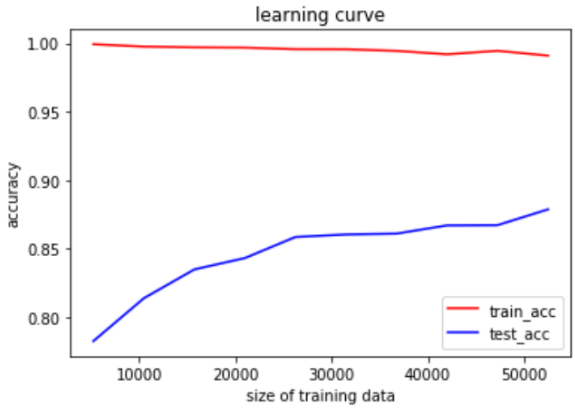


Figure 5 - learning curve

As we can observe, while the test accuracy is always close to 100%, the test accuracy keeps increasing from 80% to 87% with the increasing size of the training set. According to the curve, we believe the model can still be improved with more training data.

Then we fixed the size of training data as maximum 52500, check how the max_depth controls the overfitting by pruning.

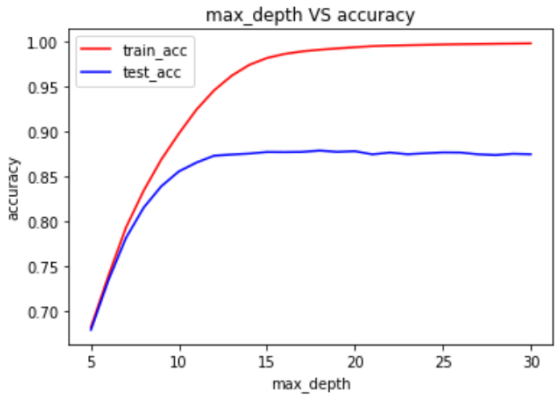


Figure 6 - max_depth VS accuracy

The results show that after max_depth is large than 15, the training accuracy keeps increasing and the test accuracy keeps flat. Also, the best test accuracy 0.879 happens with max_depth as 18. All those can show the model may already overfit when max_depth is greater than 20.

In summary, the MNIST data is much bigger in both data size and feature size. Also, the mapping function may be more complicated with more dimensions (28*28) and more output categories (10). So, there are an obvious gap between the train accuracy and test accuracy. A possible reason could be overfitting, which require us to take additional measure to achieve a better test accuracy.

Neural Network

1. Algorithm Details

Multi-layer Perceptron (MLP) is a supervised learning algorithm that learns a mapping function by training on a dataset. Given a set of features and a target, it can learn a non-linear function approximator for either classification or regression. It is different from logistic regression, in that between the input and the output layer, there can be one or more non-linear layers, called hidden layers. Figure 7 shows a one hidden layer MLP with scalar output.

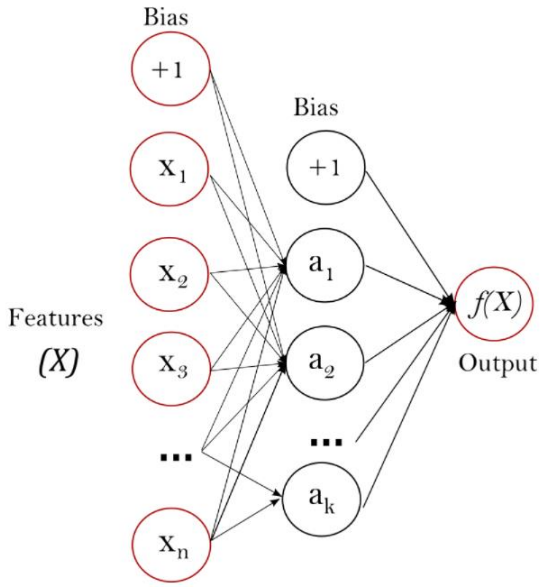


Figure 7 - One Hidden Layer MLP

Some common parameters we can adjust when training the model.

- hidden_layer_sizes: tuple, length = n_layers - 2, default=(100,)
  The ith element represents the number of neurons in the ith hidden layer.

- activation: {'identity', 'logistic', 'tanh', 'relu'} - activation function for the hidden layer.
  'identity', no-op activation, useful to implement linear bottleneck, returns f(x) = x
  'logistic', the logistic sigmoid function, returns f(x) = 1 / (1 + exp(-x)).

'tanh', the hyperbolic tan function, returns f(x) = tanh(x).
'relu', the rectified linear unit function, returns f(x) = max(0, x)

- max_iter: int, default=200
  Maximum number of iterations. The solver iterates until convergence (determined by 'tol') or this number of iterations.

For the first parameter, it decides the depth and width of the neural network.

For the second parameter, it decides the activation function in the NN model. I usually prefer to use 'relu', because the constant gradient of ReLUs both help to avoid the gradient to vanish and results in faster learning.

The last parameter. It controls the balance between underfitting and overfitting by setting the stop criteria.

2. Implementation in task 1

We split the 150-instance dataset into training set (112 - 75%) and testing set (38 - 25%).

Fixed parameters: { hidden_layer_sizes = (100), activation = 'relu' }

We record the learning curve:

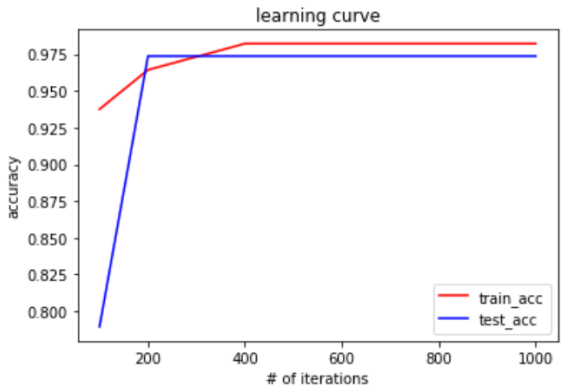


Figure 8 - learning curve

As we can observe, the training accuracy is always 100% which means the decision tree can perfectly predict the training data regardless of the size of training data. On the other hand, if the size of training data is smaller than 20, the test accuracy is much lower. And when the size of training data is bigger than 40, the test accuracy keeps as ~97%.

3. Implementation in task 2

We split the 70000-instance dataset into training set (52500 - 75%) and testing set (17500 - 25%).

Fixed parameters: { hidden_layer_sizes = (100), activation = 'relu', max_iter = 1000 }

We record the learning curve with training size:

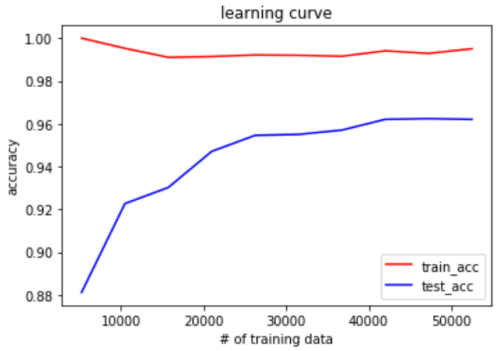


Figure 9 NN model learning curve

Then we fixed the size of training data as maximum 52500 and maximum iteration as 1000, check how the size of the NN model influence the test accuracy.

Table 1 - accuracy comparison

hidden_layer_sizes	(10)	(100)	(10,10)	(100,100)
Train accuracy	0.680	0.995	0.908	0.996
Test accuracy	0.678	0.962	0.888	0.971

As the result shows, the deeper and wider NN model with (100,100) gives us the best accuracy both in training data and test data.

AdaBoost Tree

1. Algorithm Details

AdaBoost, short for Adaptive Boosting, is a statistical classification meta-algorithm. It can be used in conjunction with many other types of learning algorithms to improve performance. The output of the other learning algorithms ('weak learners') is combined into a weighted sum that represents the final output of the boosted classifier. AdaBoost is adaptive in the sense that subsequent weak learners are tweaked in favor of those instances misclassified by previous classifiers. In some problems it can be less susceptible to the overfitting problem than other learning algorithms. The individual learners can be weak, but as long as the performance of each one is slightly better than random guessing, the final model can be proven to converge to a strong learner. AdaBoost (with decision trees as the weak learners) is often referred to as the best out-of-the-box classifier. When used with decision tree learning, information gathered at each stage of the AdaBoost algorithm about the relative 'hardness' of each training sample is fed into the tree growing algorithm such that later trees tend to focus on harder-to-classify examples.

Some common parameters we can adjust when training this model:

- base_estimator: The base estimator from which the boosted ensemble is built.
- n_estimators: The maximum number of estimators at which boosting is terminated. In case of perfect fit, the learning procedure is stopped early.
- learning_rate: Weight applied to each classifier at each boosting iteration. A higher learning rate increases the contribution of each classifier. There is a trade-off between the learning_rate and n_estimators parameters.

For the first parameter, we will borrow the best decision tree from the previous exploration.

For the second parameter, we will set the default as n_estimators = 50.

For the third parameter, we will set the learning_rate = 1.

2. Implementation in task 1

We split the 150-instance dataset into training set (112 - 75%) and testing set (38- 25%).

Fixed parameters: { base_estimator = DecisionTreeClassifier, n_estimators = 50, learning_rate = 1.0}

We record the learning curve with different training size:

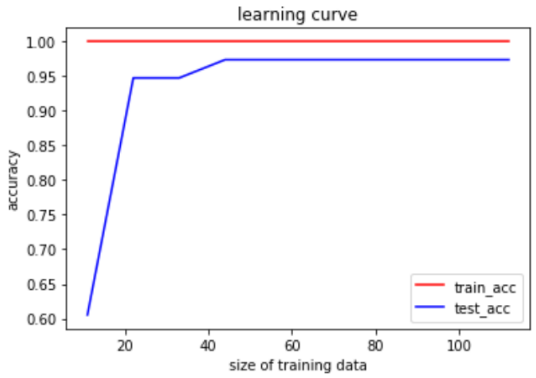


Figure 10 - AdaBoost Tree learning rate

3. Implementation in task 2

We split the 70000-instance dataset into training set (52500 - 75%) and testing set (17500 - 25%).

Fixed parameters: { base_estimator = DecisionTreeClassifier(max_depth = 18), n_estimators = 50, learning_rate = 1.0}

We record the learning curve with different training size:

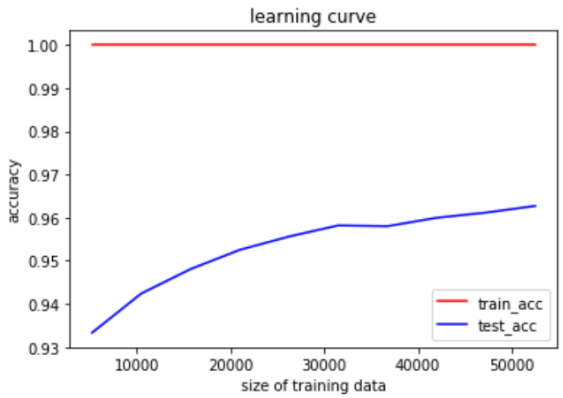


Figure 11 AdaBoost Tree learning rate

Then we fixed the size of training data as maximum 52500, check how the n_estimators influence the test accuracy.

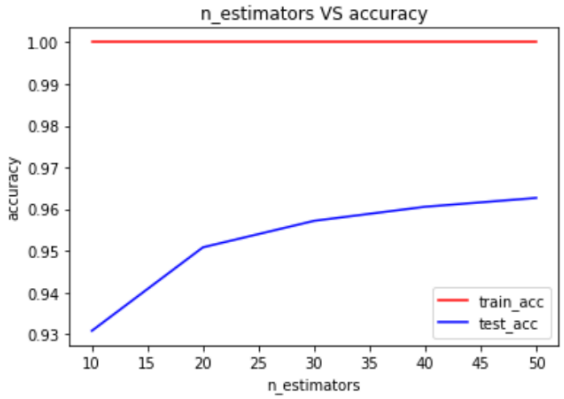


Figure 12 n_estimators VS accuracy

Intuitively, with the n_estimators increased from 10 to 50, the test accuracy also keeps increasing. And the curve become flat after n_extimators >= 30.

SVM

1. Algorithm Details

Support vector machines (SVMs) are a set of supervised learning methods used for classification, regression.

The advantages of support vector machines are:

- Effective in high dimensional spaces.
- Still effective in cases where number of dimensions is greater than the number of samples.

- Uses a subset of training points in the decision function (called support vectors), so it is also memory efficient.
- Versatile: different Kernel functions can be specified for the decision function. Common kernels are provided, but it is also possible to specify custom kernels.

The common parameters we can adjust when training the model:

- C: Regularization parameter. The strength of the regularization is inversely proportional to C. Must be strictly positive. The penalty is a squared l2 penalty.

- Kernel: {'linear', 'poly', 'rbf', 'sigmoid', 'precomputed'}, Specifies the kernel type to be used in the algorithm. It must be one of 'linear', 'poly', 'rbf', 'sigmoid', 'precomputed' or a callable.

2. Implementation in task 1

We split the 150-instance dataset into training set (112 - 75%) and testing set (38 - 25%).

Fixed parameters: { C = 1.0 }

We record the results for different kernels:

Table 2 SVM with different kernels

Kernel	Linear	Rbf	Sigmoid
Test accuracy	0.982	0.938	0.696
Train accuracy	0.974	0.921	0.579

3. Implementation in task 2

We split the 70000-instance dataset into training set (52500 - 75%) and testing set (17500 - 25%).

Fixed parameters: { C = 1.0 }

We record the results for both 'linear' and 'rbf' kernel:

Table 3 SVM with different kernels

Kernel	Linear	Rbf
Test accuracy	0.973	0.942
Train accuracy	0.932	0.940

KNN

1. Algorithm Details

In statistics, the k-nearest neighbors algorithm (k-NN) is a non-parametric classification method. It is used for classification and regression. In both cases, the input consists of the k closest training examples in data set. In k-NN classification, the output is a class membership. An object

is classified by a plurality vote of its neighbors, with the object being assigned to the class most common among its k nearest neighbors (k is a positive integer, typically small). If k = 1, then the object is simply assigned to the class of that single nearest neighbor.

k-NN is a type of classification where the function is only approximated locally and all computation is deferred until function evaluation. Since this algorithm relies on distance for classification, if the features represent different physical units or come in vastly different scales then normalizing the training data can improve its accuracy dramatically.

A useful technique can be to assign weights to the contributions of the neighbors, so that the nearer neighbors contribute more to the average than the more distant ones. For example, a common weighting scheme consists in giving each neighbor a weight of 1/d, where d is the distance to the neighbor.

The neighbors are taken from a set of objects for which the class is known. This can be thought of as the training set for the algorithm, though no explicit training step is required.

A peculiarity of the k-NN algorithm is that it is sensitive to the local structure of the data.

Some common parameters we can adjust when training this model:

- n_neighbors: Number of neighbors to use by default for kneighbors queries.

- Weights: {'uniform', 'distance'} or callable. Weight function used in prediction. Possible values:
  'uniform': uniform weights. All points in each neighborhood are weighted equally
  'distance': weight points by the inverse of their distance. in this case, closer neighbors of a query point will have a greater influence than neighbors which are further away.

2. Implementation in task 1

We split the 150-instance dataset into training set (112 - 75%) and testing set (38 - 25%).

Fixed parameters: { n_neighbors = 3}

We got a train accuracy 0.964 and a test accuracy 0.974.

3. Implementation in task 2

We split the 70000-instance dataset into training set (52500 - 75%) and testing set (17500 - 25%).

We record the results by adjust the n_neighbors from 1 to 5, and we get the highest train accuracy 97% with n_neighbors as 3.

Results Comparison

For task 1, all algorithms are in a good shape with almost 100% test accuracy and 97% test accuracy without additional tunning.

For task 2, we mainly compare the best test accuracy for above 5 datasets as following.

Table 4 - Best test accuracy for different models in MNIST dataset

	Decision Tree	Neural Network	AdaBoost Tree	SVM	KNN
TEST ACCURACY	88%	96%	96%	94%	97%

Summary

- Compare with Single Decision Tree, ensemble model using AdaBoost Tree improves the accuracy a lot in MNITS handwriting detection task.
- In each algorithm, there are some parameters which can help to control the balance between underfitting and overfitting. Some examples will be the tree depth in Decision Tree, C value in SVM and so on.