CS-GY 6923 Final Project Report

Mengxi Wu (mw4355@nyu.edu)

Qi Yin (qy652@nyu.edu)

Synopsis：

In this project, we do three extensions on decision tree, logistic regression and neural network.

Firstly, we chose decision tree to do extension:

We changed single decision tree into random forest.

1.Test the improvement with Sklearn:

```
single_tree_sklearn = DecisionTreeClassifier(criterion="entropy")
single_tree_sklearn.fit(X_train, y_train)
sklearn_tree_predictions = single_tree_sklearn.predict(X_test)
print("sklearn Single Tree Accuracy: ", evaluate(sklearn_tree_predictions, y_test))

forest_sklearn = RandomForestClassifier(n_estimators=20)
forest_sklearn.fit(X_train, y_train)
sklearn_forest_predictions = forest_sklearn.predict(X_test)
print("sklearn Random Forest Accuracy: ", evaluate(sklearn_forest_predictions, y_test))
```

```
sklearn Single Tree Accuracy:  0.8386648122392212
sklearn Random Forest Accuracy:  0.9541029207232267
```

The dataset we use to test the effects is handwritten digits dataset.

2.Modify our implementation of decision tree

We modify three places of decision tree. First one, we change criterion entropy into gini index. We implement gini index instead of entropy because when we test two criterions on the handwritten digits dataset with sklearn, decision tree with gini index performances better. We also change our preprocess function. We first find the maximum and minimum in the training data, then we divide the data range into different intervals. Each interval correspond to an integer (0,1,2…). If the data falls into the interval, we reassign the data to corresponding integer of that interval. The aim to do this is to change continuous data into categorical data.

The third modification is to change single decision tree into random forest. We use 20 trees in the implementation. The reason to choose 20 because we try different number of tree and 20 works out the best on handwritten digit dataset. Unlike what mentions in the lecture slide, we choose 2/3 of features instead of square root number of features. This is because the square root of features is too small; there is too much feature loss and the classifer won't performance well and the accuracies are very unstable. For each tree, we use 2/3 of training examples to build the tree.

The reason why random forest works better than a single decision tree is that it reduces variance meanwhile limits overfitting without increasing errors due to bias. Random forest reduces variance

through training on different samples of the data. Though for each tree we only randomly choose a subset of features to use, the errors due to bias won't increase significantly, since if we use enough trees in the forest, all the features can be included and considered.

The accuracies of our implementation of decision tree and random forest on handwritten digit dataset are listed below:

```
data_range = (X_train.min(0), X_train.max(0))
single_tree = DecisionTree(3, data_range)
single_tree.train(X_train, y_train)
predictions = single_tree.predict(X_test)
print("Single Tree Accuracy: ", evaluate(predictions, y_test))

random_forest(X_train, y_train, X_test, y_test, 3, 20)
```

```
Single Tree Accuracy:  0.6898470097357441
Random Forest Accuracy:  0.885952712100139
```

3.Test on other dataset

We choose UCI spambase dataset to test our implementation. The description of the dataset is listed below:

Data Set Characteristics:	Multivariate	**Number of Instances:**	4601	**Area:**	Computer
Attribute Characteristics:	Integer, Real	**Number of Attributes:**	57	**Date Donated**	1999-07-01
Associated Tasks:	Classification	**Missing Values?**	Yes	**Number of Web Hits:**	464269

Like handwritten digit dataset, this dataset is continuous, we use the same method mentioned above to preprocess this data. We choose the number of intervals to be 10. We also use 20 trees in the forest. The accuracies is listed below:

```
data_range = (X_train.min(0), X_train.max(0))
single_tree = DecisionTree(10, data_range)
single_tree.train(X_train, y_train)
predictions = single_tree.predict(X_test)
print("Single Tree Accuracy: ", evaluate(predictions, y_test))

random_forest(X_train, y_train, X_test, y_test, 10, 20)
```

```
Single Tree Accuracy:  0.766875
Random Forest Accuracy:  0.861875
```

Secondly, we chose logistic regression to do extension:

We changed Logistic Regression into Ridge Logistic Regression.

1.Test the improvement with Sklearn:

```
# experiment in Sklearn:
from sklearn.linear_model import LogisticRegression
logreg = LogisticRegression(penalty ='none',class_weight= 'balanced', random_state=0, solver='newton-cg', multi_class='ovr')
logreg.fit(X_train, y_train)
prepro = logreg.predict_proba(X_test)
acc = logreg.score(X_test,y_test)
print("None: ")
print (acc)
logreg = LogisticRegression(penalty ='l2',class_weight= 'balanced', random_state=0, solver='newton-cg', multi_class='ovr')
logreg.fit(X_train, y_train)
prepro = logreg.predict_proba(X_test)
acc = logreg.score(X_test,y_test)
print ("L2:")
print (acc)
```

```
None:
0.9304589707927677
L2:
0.9652294853963839
```

2. Modify our implementation of logistic regression

Since the Logistic regression is a binary classification, while the MNST has 10 classes. We tried two ways to modify the dataset, the first one is to combine class 0-4 as the first class, and to combine 5-9 as the second class. But the result is not good, even in Sklearn's logistic classifier the accuracy just improves 0.02%. We thought that the reason is that this kind of combination decrease the complexity of the dataset, so the overfitting problem is not that serious. As a result, adding penalty term isn't that useful.

So, we tried the second solution. We changed logistic regression into a multiclass classification. We generated 10 group of training data from the original one, and then trained 10 classifiers corresponding to each class. When doing classification, we just put the test data into 10 classifiers, and choose the one with highest score.

The result becomes better.

```
The logistic regression precision is: 0.8706536856745479

In ridge regression,C = 0.6, the precision is: 0.8803894297635605
In ridge regression,C = 0.65, the precision is: 0.8803894297635605
In ridge regression,C = 0.7, the precision is: 0.8803894297635605
In ridge regression,C = 0.75, the precision is: 0.8803894297635605
In ridge regression,C = 0.8, the precision is: 0.8803894297635605
```

It shows that after adding the penalty term to the objective function, the accuracy improved.The reason why adding penalty term can improve accuracy is that it can help to prevent overfitting problem. In logistic regression, we maximize the likelihood of the dataset to get the coefficient. Since we are trying our best to fit the training dataset, it leads to large w, and may overfitting of the data. In an overfitted model, although training error becomes almost 0, it lost some ability to correctly predict new data (lower bias, but higher variance). So, it performs worse on test dataset. But added penalty term, we tend to avoid very large w, we shrink coefficients to some extent. So that, we protect our model from outliers in the

training dataset that might skew coefficients drastically, and we maintain fewer variables(some features not that matters will be really small). As a result, it prevent overfitting. From the perspective of error, we trade off some bias to get lower variance, and lower variance estimators tend to overfit less.

3. Test on another dataset

The second data set we chose is “Titanic dataset” from Kaggle, the content of the dataset is:

Variable	Definition	Key
survival	Survival	0 = No, 1 = Yes
pclass	Ticket class	1 = 1st, 2 = 2nd, 3 = 3rd
sex	Sex	
Age	Age in years	
sibsp	# of siblings / spouses aboard the Titanic	
parch	# of parents / children aboard the Titanic	
ticket	Ticket number	
fare	Passenger fare	
cabin	Cabin number	
embarked	Port of Embarkation	C = Cherbourg, Q = Queenstown, S = Southampton

To propcess the data, we delete the data that with null value. Besides, since the goal is to predict whether the passenger will survive, we delete some irrelevance features, such as “passenger ID”, ”Name”, “Ticket”, “Cabin”. And then since Sec and Embarked are out-of-oder categorical variable, we encoded them. Finally, in order to ensure that our model can converge soon, we use “standardScaler” to normalize our data.

The result is as bellow:

```
The logistic regression precision is: 0.7616822429906542
C = 0.04, The Ridge regression precision is: 0.7710280373831776
C = 0.05, The Ridge regression precision is: 0.7757009345794392
C = 0.06, The Ridge regression precision is: 0.7710280373831776
C = 0.03, The Ridge regression precision is: 0.7710280373831776
```

Thirdly, we chose neural network to do the exention:

We change the optimize function, use Adaptive Gradient Algorithm instead of Batch Gradient Decent.

1.Test the improvement with Tensorflow:

This extension we haven't tried on tensorflow but it does enhance the performance practically.

2. Modify our implementation of neural network

There are two places changed in neural network. First, we choose to use sigmoid function as our activation function. Second, we choose Ada Grad instead of Batch Gradient Acent. The modifications are shown below:

Initialize hw and hb for Ada Grad

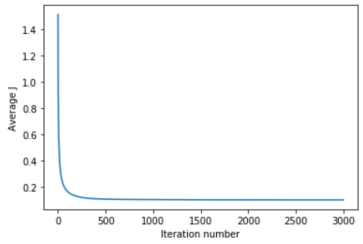
```
def init_h_w_b_values(nn_structure):
    h_w = {}
    h_b = {}
    for l in range(1, len(nn_structure)):
        h_w[l] = np.zeros((nn_structure[l], nn_structure[l-1]))
        h_b[l] = np.zeros((nn_structure[l],))
    return h_w, h_b
```

```
for l in range(len(nn_structure) - 1, 0, -1):
    h_w[l] += tri_W[l] * tri_W[l]
    h_b[l] += tri_b[l] * tri_b[l]

    W[l] += -alpha * (tri_W[l]/(np.sqrt(h_w[l]) + 1e-7) + lamb/2*W[l])
    b[l] += -alpha * (tri_b[l]/(np.sqrt(h_b[l]) + 1e-7))
```

Adagrad works better than gradient decent. It allows to perform larger updates for parameters associated wi th infrequent occurring features and smaller update for parameters associated with frequent occurring featur es. It fits well for sparse data. It also eliminates the need to manually tune the learning rate. For batch gr adient descent, the learning rate for each parameter is same. In Adagrad, each parameter has different learn ing rate and the learning rate changes in every step. We divide the sum of the square of the past gradient s. Since the term is positive and it continues to increase, the learning rate decreases in each step. The per formances of Batch gradient descent and Adagrad on handwritten digit dataset are listed below:

Plotting the learning curve for handwritten digits data for Adagrad

```
# plot the avg_cost_func
plt.plot(avg_cost_func)
plt.ylabel('Average J')
plt.xlabel('Iteration number')
plt.show()
```



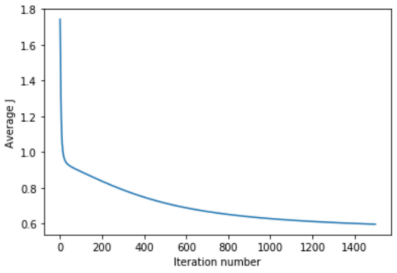
Prediction accuracy for handwritten digits data for Adagrad

```
# get the prediction accuracy and print
y_pred = predict_y(W, b, X_test, 3)
print('Prediction accuracy is {}%'.format(accuracy_score(y_test, y_pred) * 100))
```

```
Prediction accuracy is 97.63560500695411%
```

Ploting the learning curve for handwritten digits data for BGD

```
nn_structure = [64, 30, 10]
al = 0.25
la = 0.01
activation = 1
W, b, avg_cost_func = train_nn(nn_structure, X_train, y_v_train, 1500,al,la,activation)
plt.plot(avg_cost_func)
plt.ylabel('Average J')
plt.xlabel('Iteration number')
plt.show()
```



Ploting the learning curve for handwritten digits data for BGD

```
y_pred = predict_y(W, b, X_test, 3,activation)
print('Prediction accuracy is {}%'.format(accuracy_score(y_test, y_pred) * 100))
```

```
Prediction accuracy is 92.35048678720446%
```

From the results above, Adagrad has accuracy 97.64%. Batch gradient descent has accuracy 92.35%. We can see Adagrad leads to higher accuracy.

3. Test on another dataset

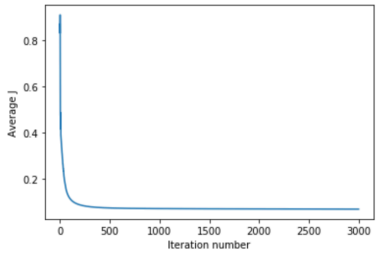
The second dataset we choose is Iris dataset. The description of this dataset is:

Data Set Characteristics:	Multivariate	**Number of Instances:**	150	**Area:**	Life
Attribute Characteristics:	Real	**Number of Attributes:**	4	**Date Donated**	1988-07-01
Associated Tasks:	Classification	**Missing Values?**	No	**Number of Web Hits:**	3014527

The performance of Adagrad on this dataset is:

Plotting the learning curve for iris data for Adagrad

```
# plot the avg_cost_func
plt.plot(avg_cost_func)
plt.ylabel('Average J')
plt.xlabel('Iteration number')
plt.show()
```



Prediction accuracy for iris data for Adagrad

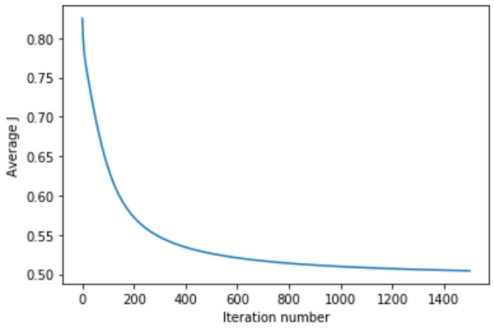
```
# get the prediction accuracy and print
y_pred = predict_y(W, b, X_test, 3)
print('Prediction accuracy is {}%'.format(accuracy_score(y_test, y_pred) * 100))
```

```
Prediction accuracy is 96.66666666666667%
```

The performance of Batch Gradient Descent on this dataset is:

Ploting the learning curve for iris data for BGD

```
nn_structure = [4, 30, 3]
al = 0.25
la = 0.01
activation = 1
W, b, avg_cost_func = train_nn(nn_structure, X_train, y_v_train, 1500,al,la,activation)
plt.plot(avg_cost_func)
plt.ylabel('Average J')
plt.xlabel('Iteration number')
plt.show()
```



Ploting the learning curve for iris data for BGD

```
y_pred = predict_y(W, b, X_test, 3,activation)
print('Prediction accuracy is {}%'.format(accuracy_score(y_test, y_pred) * 100))
```

```
Prediction accuracy is 91.66666666666666%
```

From the results above, on Iris dataset, Adagrad also leads to higher accuracy.

Decision Tree Extension

In [63]:
```
import sklearn
import pandas as pd
import numpy as np
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
import pandas as pd
import math
import random
```

Decision Tree Implementation with Gini Index

In [64]:
```
class DecisionTree:

    def __init__(self, nbins, data_range):
        # Decision tree state here
        # Feel free to add methods
        self.bin_size = nbins
        self.range = data_range

    def preprocess(self, data):
        # Our dataset only has continuous data
        norm_data = np.clip((data - self.range[0]) / (self.range[1] - self.range[0]), 0, 1)
        categorical_data = np.floor(self.bin_size * norm_data).astype(int)
        return categorical_data

    def train(self, X, y):
        # training logic here
        # input is array of features and labels
        categorical_data = self.preprocess(X)
        feature_names = []
        for i in range(categorical_data.shape[1]):
            feature_names.append(i)
        X = list(categorical_data)
        y = list(y)
        self.tree = self.build_tree(X, y, [],feature_names)

    def predict(self, X):
        # Run model here
        # Return array of predictions where there is one prediction for each set of features
        categorical_data = self.preprocess(X)
        predict_results = []
        for x in categorical_data:
            label = self.get_label(self.tree, x)
            predict_results.append(label)
        result = np.array(predict_results)
        return result

    def gini(self, labels):
        class_count = {}
        for label in labels:
            if label not in class_count.keys():
                class_count[label] = 1
            else:
                class_count[label] += 1
        giniValue = 0
        for label in class_count.keys():
            prob = class_count[label] * 1.0 / len(labels)
            giniValue += prob * prob
        return 1 - giniValue

    def get_attribute_label(self, feature_column, attribute, labels):
        attri_labels = []
        for i in range(len(labels)):
            if feature_column[i] == attribute:
                attri_labels.append(labels[i])
        return attri_labels

    def get_attribute_rows(self, feature_column, feature_selected, attribute, features):
        new_features = []
        for i in range(len(features)):
            if feature_column[i] == attribute:
                new_features.append(features[i])
        if new_features != []:
            new_features = np.delete(new_features, feature_selected, 1)
        return new_features

    def get_gini(self, feature_column, labels):
        attributes = set(feature_column)
        feature_gini= 0
        for attribute in attributes:
            attri_labels = self.get_attribute_label(feature_column, attribute, labels)
            attri_count = feature_column.count(attribute)
            feature_gini += attri_count/len(feature_column) * self.gini(attri_labels)
        return feature_gini

    def get_majority(self, labels):
        class_count = {}
        for label in labels:
            if label not in class_count.keys():
                class_count[label] = 1
            else:
                class_count[label] += 1
        mostVote = -1000
        labelSelected = -1
        for label in class_count.keys():
            if class_count[label] > mostVote:
                mostVote = class_count[label]
                labelSelected = label
        return labelSelected

    def is_labels_all_same(self, labels):
        counter = labels.count(labels[0])
        if counter == len(labels):
            return True
        return False

    def is_row_all_same(self, X):
        return np.equal(X[0], X).all()

    def build_tree(self, X, y, parent_y, feature_names):
        if len(y) == 0:
            return self.get_majority(parent_y)

        if self.is_labels_all_same(y):
            return y[0]

        if len(feature_names) == 1 or self.is_row_all_same(X):
            return self.get_majority(y)

        min_gini = 1000
        selected = -1
        for i in range(len(feature_names)):
            feature_column = [x[i] for x in X]
            gini = self.get_gini(feature_column, y)
            if gini < min_gini:
                min_gini = gini
                selected = i

        best_feature = feature_names[selected]
        feature_column = [x[selected] for x in X]
        feature_names.remove(feature_names[selected])

        tree = {best_feature: {}}
        attributes = set(feature_column)

        for attribute in attributes:
            new_X = self.get_attribute_rows(feature_column, selected, attribute, X)
            new_y = self.get_attribute_label(feature_column, attribute, y)
            sub_feature_names = feature_names[:]
            tree[best_feature][attribute] = self.build_tree(new_X, new_y, y, sub_feature_names)
        return tree

    def get_label(self, tree, x):
        key_list = list(tree.keys())
        feature = key_list[0]
        predict = -1000
        for key in tree[feature].keys():
            if x[feature] == key:
                if type(tree[feature][key]).__name__ == 'dict':
                    predict = self.get_label(tree[feature][key], x)
                else:
                    predict = tree[feature][key]
        return predict
```

Random Forest Implementation

In [65]:
```
def random_forest(X_train, y_train, X_test, y_test, num_bin, num_tree):
    num_feature = math.floor(2/3*len(X_train[0]))
    num_train = math.floor(2/3*len(X_train))
    feature_names = []
    train_used = []
    for i in range(len(X_train[0])):
        feature_names.append(i)

    for i in range(len(X_train)):
        train_used.append(i)

    results = []
    for i in range(num_tree):
        random.shuffle(feature_names)
        random.shuffle(train_used)
        feature_choose = feature_names[:num_feature]
        train_examples = train_used[:num_train]
        new_X_train = X_train[:, feature_choose]
        new_X_train = new_X_train[train_examples, :]
        new_y_train = y_train[train_examples]
        data_range = (new_X_train.min(0), new_X_train.max(0))
        tree = DecisionTree(num_bin, data_range)
        tree.train(new_X_train, new_y_train)
        new_X_test = X_test[:, feature_choose]
        predictions = tree.predict(new_X_test)
        results.append(predictions)

    results = np.array(results)
    solutions = []
    for i in range(len(results[0])):
        solution = get_majority(results[:, i])
        solutions.append(solution)

    solutions = np.array(solutions)
    y_labels = np.array(y_test)
    print("Random Forest Accuracy: ", evaluate(solutions, y_test))
```

Evaluation Function

In [66]:
```
def evaluate(solutions, real):
    if(solutions.shape != real.shape):
        raise ValueError("Output is wrong shape.")
    predictions = np.array(solutions)
    labels = np.array(real)
    return (predictions == labels).sum() / float(labels.size)

def get_majority(labels):
    class_count = {}
    for label in labels:
        if label not in class_count.keys():
            class_count[label] = 1
        else:
            class_count[label] += 1
    mostVote = -1000
    labelSelected = -1
    for label in class_count.keys():
        if class_count[label] > mostVote:
            mostVote = class_count[label]
            labelSelected = label
    return labelSelected
```

Test on Handwritten digit data

In [76]:
```
digits=load_digits()
X = digits.data
y = digits.target
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.4)
print(X_train.shape)
print(y_train.shape)
```

```
(1078, 64)
(1078,)
```

(1) Compare effects between single decision tree and random forest on sklearn

In [77]:
```
single_tree_sklearn = DecisionTreeClassifier(criterion="entropy")
single_tree_sklearn.fit(X_train, y_train)
sklearn_tree_predictions = single_tree_sklearn.predict(X_test)
print("Sklearn Single Tree Accuracy: ", evaluate(sklearn_tree_predictions, y_test))

forest_sklearn = RandomForestClassifier(n_estimators=20)
forest_sklearn.fit(X_train, y_train)
sklearn_forest_predictions = forest_sklearn.predict(X_test)
print("Sklearn Random Forest Accuracy: ", evaluate(sklearn_forest_predictions, y_test))
```

```
sklearn Single Tree Accuracy:  0.8442280945757997
sklearn Random Forest Accuracy:  0.9541029207232267
```

(2) Compare effects between single decision tree and random forest on our implementation

In [70]:
```
data_range = (X_train.min(0), X_train.max(0))
single_tree = DecisionTree(3, data_range)
single_tree.train(X_train, y_train)
predictions = single_tree.predict(X_test)
print("Single Tree Accuracy: ", evaluate(predictions, y_test))

random_forest(X_train, y_train, X_test, y_test, 3, 20)
```

```
/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:11: RuntimeWarning: divide by zero encountered in true_divide
  # This is added back by InteractiveShellApp.init_path()
/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:11: RuntimeWarning: invalid value encountered in true_divide
  # This is added back by InteractiveShellApp.init_path()

Single Tree Accuracy:  0.6898470097357441
Random Forest Accuracy:  0.8859527121001139
```

Test on Spambase data

In [71]:
```
data = pd.read_csv("spambase.csv", delimiter=',')
data = data.sample(frac=1).reset_index(drop=True)
X_train = data.loc[:3000, "word1":"word57"].values
y_train = data.loc[:3000, "labels"].values
X_test = data.loc[3001:, "word1":"word57"].values
y_test = data.loc[3001:, "labels"].values
```

In [72]:
```
data_range = (X_train.min(0), X_train.max(0))
single_tree = DecisionTree(10, data_range)
single_tree.train(X_train, y_train)
predictions = single_tree.predict(X_test)
print("Single Tree Accuracy: ", evaluate(predictions, y_test))

random_forest(X_train, y_train, X_test, y_test, 10, 20)
```

```
Single Tree Accuracy:  0.766875
Random Forest Accuracy:  0.861875
```

Logistic Regression Extension

In [2]:
```
from sklearn.datasets import load_digits # The MNIST data set is in scikit learn data set
from sklearn.preprocessing import StandardScaler  # It is important in neural networks to scale the
 date
from sklearn.model_selection import train_test_split  # The standard - train/test to prevent overfit
ting and choose hyperparameters
from sklearn.metrics import accuracy_score #
import numpy as np
import numpy.random as r # We will randomly initialize our weights
import matplotlib.pyplot as plt

# load dataset
digits=load_digits()
X = digits.data
y = digits.target

# scalar dataset
X_scale = StandardScaler()
X = X_scale.fit_transform(digits.data)

#split original dataset
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.4)
```

Create binary dataset

In [13]:
```
# rule label:0-4 count as 0, label:5-9 count as 1
y_binary = np.zeros(len(y))
for i in range(len(y)):
    if  y[i] == 0 or y[i] == 1 or y[i] == 2 or y[i] == 3 or y[i] == 4:
        y_binary[i] = 0
    else:
        y_binary[i] =1
y= np.ravel(y)

#Split the binary data set
X_binary_train, X_binary_test, y_binary_train, y_binary_test = train_test_split(X, y_binary, test_si
ze=0.4)
```

Create data set for muti classification of logistic regression

In [14]:
```
new_y_train_list =[] # 0-9

for j in range(10):
    new_y = np.zeros(len(y_train))
    for i in range(len(y_train)):
        if  y_train[i] == j:
            new_y[i] = 1
        else:
            new_y[i] = 0
    new_y_train_list.append(new_y)
new_y_train_list = np.mat(new_y_train_list)
print("The original train set is X:{}, y:{}".format(X_train.shape,y_train.shape))
print("We create 9 trainning data set according to class 0-9.")
print("The new train set is New y List: {}".format(new_y_train_list.shape))
print("The original y is: {}".format(y_train))
print("The 0 class: y:{}".format(new_y_train_list[0]))
print("The 1 class: y:{}".format(new_y_train_list[1]))
print("The 2 class: y:{}".format(new_y_train_list[2]))
print("The 3 class: y:{}".format(new_y_train_list[3]))
print("The 4 class: y:{}".format(new_y_train_list[4]))
print("The 5 class: y:{}".format(new_y_train_list[5]))
print("The 6 class: y:{}".format(new_y_train_list[6]))
print("The 7 class: y:{}".format(new_y_train_list[7]))
print("The 8 class: y:{}".format(new_y_train_list[8]))
print("The 9 class: y:{}".format(new_y_train_list[9]))
```

```
The original train set is X:(1078, 64), y:(1078,)
We create 9 trainning data set according to class 0-9.
The new train set is New y List: (10, 1078)
The original y is: [9 4 2 ... 1 2 3]
The 0 class: y:[[0. 0. 0. ... 0. 0. 0.]]
The 1 class: y:[[0. 0. 0. ... 1. 0. 0.]]
The 2 class: y:[[0. 0. 1. ... 0. 1. 0.]]
The 3 class: y:[[0. 0. 0. ... 0. 0. 1.]]
The 4 class: y:[[0. 1. 0. ... 0. 0. 0.]]
The 5 class: y:[[0. 0. 0. ... 0. 0. 0.]]
The 6 class: y:[[0. 0. 0. ... 0. 0. 0.]]
The 7 class: y:[[0. 0. 0. ... 0. 0. 0.]]
The 8 class: y:[[0. 0. 0. ... 0. 0. 0.]]
The 9 class: y:[[1. 0. 0. ... 0. 0. 0.]]
```

Validation in Sklearn

In [15]:
```
# experiment in Sklearn:
from sklearn.linear_model import LogisticRegression
logreg = LogisticRegression(penalty ='none',class_weight= 'balanced', random_state=0, solver='newton
-cg', multi_class='ovr')
logreg.fit(X_train, y_train)
prepro = logreg.predict_proba(X_test)
acc = logreg.score(X_test,y_test)
print("None: ")
print (acc)
logreg = LogisticRegression(penalty ='l2',class_weight= 'balanced', random_state=0, solver='newton-c
g', multi_class='ovr')
logreg.fit(X_train, y_train)
prepro = logreg.predict_proba(X_test)
acc = logreg.score(X_test,y_test)
print ("L2:")
print (acc)
```

```
None:
0.9360222531293463
L2:
0.9499304589707928
```

Our implementation of logistic regression

In [16]:
```
# Some params
learning_rate = 0.1
num_iters = 3000 # The number of iteratins to run the gradient ascent algorithm

# logistc regression
def sigmoid(z):
    return 1/(1+np.exp(-z))

# Initialize parameters w
w = np.zeros((X_train.shape[1], 1))

def hypothesis(X , w):
    return 1/(1+np.exp(-np.dot(X,w)))
yhat = hypothesis(X_train, w)

def log_likelihood(X ,y , w ):
    hx = hypothesis(X , w)
    log_likelihood =0
    for i in range(X.shape[0]):
        if y[i] == 0:
            if hx[i] ==1:
                continue
            log_likelihood = log_likelihood + np.log(1-hx[i])
        else:
            if hx[i] ==0:
                continue
            log_likelihood = log_likelihood + np.log(hx[i])
    return log_likelihood

def Logistic_Regresion_Gradient_Ascent(X, y, learning_rate, num_iters):
    #initialize
    log_likelihood_values = []
    w = np.zeros((X.shape[1], 1))
    N = X.shape[0]
    #do iteration
    for i in range(num_iters):
        gradient = np.dot(X.transpose(),(y-hypothesis(X,w)))
        w = w + (learning_rate/N)*gradient
        if (i % 100) == 0:
            log_likelihood_values.append(log_likelihood(X,y,w))
    return w, log_likelihood_values

def ridge_log_likelihood(X , y, w, C = 0.65):
    hx = hypothesis(X , w)
    log_likelihood = 0
    for i in range(X.shape[0]):
        if y[i] == 0:
            if hx[i] ==1:
                continue
            log_likelihood = log_likelihood + np.log(1-hx[i])
        else:
            if hx[i] ==0:
                continue
            log_likelihood = log_likelihood + np.log(hx[i])
    reg_term = C*np.dot(w.T,w)
    log_likelihood = log_likelihood  - reg_term
    return log_likelihood

def Ridge_Regresion_Gradient_Ascent(X, y, learning_rate, num_iters, C = 0.65):
    #initialize
    ridge_log_likelihood_values = []
    w = np.zeros((X.shape[1], 1))
    N = X.shape[0]
    #do iteration
    for i in range(num_iters):
        gradient = np.dot(X.transpose(),(y - hypothesis(X,w)))
        w = w + (learning_rate/N)*gradient - learning_rate*C*w
        if (i % 100) == 0:
            ridge_log_likelihood_values.append(ridge_log_likelihood(X, y, w, C))
    return w, ridge_log_likelihood_values
```

Build multiclass model

In [17]:
```
# train n model (0,9)
w =[]
log_likelihood_values =[]
for i in range(10):
    w_new, log_likelihood_values_new = Ridge_Regresion_Gradient_Ascent(X_train, new_y_train_list[i].
transpose(), learning_rate, num_iters, 0.65)
    w.append(w_new)
    log_likelihood_values.append(log_likelihood_values_new)
```

Predict with ridge multiclass regression

In [18]:
```
#pretdict
result = []
for i in range(len(y_test)):
    hx_every_example = []
    for j in range(len(w)):
        hx = hypothesis(X_test[i],w[j])
        hx_every_example.append(np.linalg.det(hx))
    predict_class =np.argmax(hx_every_example)
    result.append(predict_class)

#caculate precision
right = 0
for i in range(len(y_test)):
    if result[i] == y_test[i]:
        right = right + 1

print("The precision is: {}".format(right/len(y_test)))
```

```
The precision is: 0.8803894297635605
```

In [19]:
```
# to find best C
for C in [0.6, 0.65, 0.7, 0.75, 0.8]:
    # train n model (0,9)
    w =[]
    log_likelihood_values =[]
    for i in range(10):
        w_new, log_likelihood_values_new = Ridge_Regresion_Gradient_Ascent(X_train, new_y_train_list
[i].transpose(), learning_rate, num_iters, C)
        w.append(w_new)
        log_likelihood_values.append(log_likelihood_values_new)
    #pretdict
    result = []
    for i in range(len(y_test)):
        hx_every_example = []
        for j in range(len(w)):
            hx = hypothesis(X_test[i],w[j])
            hx_every_example.append(np.linalg.det(hx))
        predict_class =np.argmax(hx_every_example)
        result.append(predict_class)

    #caculate precision
    right = 0
    for i in range(len(y_test)):
        if result[i] == y_test[i]:
            right = right + 1

    print("In ridge regression,C = {}, the precision is: {}".format(C, right/len(y_test)))
```

```
In ridge regression,C = 0.6, the precision is: 0.8803894297635605
In ridge regression,C = 0.65, the precision is: 0.8803894297635605
In ridge regression,C = 0.7, the precision is: 0.8803894297635605
In ridge regression,C = 0.75, the precision is: 0.8803894297635605
In ridge regression,C = 0.8, the precision is: 0.8803894297635605
```

Predict with logistic multiclass regression

In [20]:
```
# the accuracy of original one
# train n model (0,9)
w =[]
#log_likelihood_values =[]
for i in range(10):
    w_new, log_likelihood_values_new = Logistic_Regresion_Gradient_Ascent(X_train, new_y_train_list[
i].transpose(), learning_rate, num_iters)
    w.append(w_new)

#pretdict
result = []
for i in range(len(y_test)):
    hx_every_example = []
    for j in range(len(w)):
        hx = hypothesis(X_test[i],w[j])
        hx_every_example.append(np.linalg.det(hx))
    predict_class =np.argmax(hx_every_example)
    result.append(predict_class)

#caculate precision
right = 0
for i in range(len(y_test)):
    if result[i] == y_test[i]:
        right = right + 1

print("The logistic regression precision is: {}".format(right/len(y_test)))
```

```
The logistic regression precision is: 0.8706536856745479
```

Test on another dataset

In [3]:
```
#test on another dataset
import pandas as pd

data = pd.read_csv("train.csv")

data = data.drop(labels=['PassengerId', 'Name', 'Ticket', 'Cabin'], axis=1)
data = data.dropna()
data_dummy = pd.get_dummies(data[['Sex', 'Embarked']])
data_conti = pd.DataFrame(data, columns=['Survived', 'Pclass', 'Age', 'SibSp', 'Parch', 'Fare'], ind
ex=data.index)
data = data_conti.join(data_dummy)

#split data into X and y
X = data.iloc[:, 1:]
y = data.iloc[:, 0]

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=0)

# standard
stdsc = StandardScaler()
X_train_conti_std = stdsc.fit_transform(X_train[['Age', 'SibSp', 'Parch', 'Fare']])
X_test_conti_std = stdsc.fit_transform(X_test[['Age', 'SibSp', 'Parch', 'Fare']])

# change ndarray into dataframe
X_train_conti_std = pd.DataFrame(data=X_train_conti_std, columns=['Age', 'SibSp', 'Parch', 'Fare'],
index=X_train.index)
X_test_conti_std = pd.DataFrame(data=X_test_conti_std, columns=['Age', 'SibSp', 'Parch', 'Fare'], in
dex=X_test.index)

# Pclass is an ordered categorical variable
X_train_cat = X_train[['Pclass']]
X_test_cat = X_test[['Pclass']]
#   disordered encoded categorical variable
X_train_dummy = X_train[['Sex_female', 'Sex_male', 'Embarked_C', 'Embarked_Q', 'Embarked_S']]
X_test_dummy = X_test[['Sex_female', 'Sex_male', 'Embarked_C', 'Embarked_Q', 'Embarked_S']]

# linked them to the dataframe
X_train_set = [X_train_cat, X_train_conti_std, X_train_dummy]
X_test_set = [X_test_cat, X_test_conti_std, X_test_dummy]
X_train = pd.concat(X_train_set, axis=1)
X_test = pd.concat(X_test_set, axis=1)

#change back into the ndarray
X_train = X_train.values
X_test = X_test.values
y_train = y_train.values
y_test = y_test.values
```

```
y_train = np.mat(y_train)
```

In [29]:

```
# Some params
learning_rate = 0.1
num_iters = 3000 # The number of iteratins to run the gradient ascent algorithm

# train logistic regression
w1, log_likelihood_values1 = Logistic_Regresion_Gradient_Ascent(X_train, y_train.transpose(), learni
ng_rate, num_iters)
#pretdict
result = []
for i in range(len(y_test)):
    hx = hypothesis(X_test[i],w1)
    if hx > 0.5:
        result.append(1)
    else:
        result.append(0)

#caculate precision
right = 0
for i in range(len(y_test)):
    if result[i] == y_test[i]:
        right = right + 1

print("The logistic regression precision is: {}".format(right/len(y_test)))



# train ridge regression
for C in [0.04, 0.05,0.06, 0.03]:
    w2, log_likelihood_values2= Ridge_Regresion_Gradient_Ascent(X_train, y_train.transpose(), learni
ng_rate, num_iters, C)
    #pretdict
    result = []
    for i in range(len(y_test)):
        hx = hypothesis(X_test[i],w2)
        if hx > 0.5:
            result.append(1)
        else:
            result.append(0)

    #caculate precision
    right = 0
    for i in range(len(y_test)):
        if result[i] == y_test[i]:
            right = right + 1

    print("C = {}, The Ridge regression precision is: {}".format(C, right/len(y_test)))
```

```
The logistic regression precision is: 0.7616822429906542
C = 0.04, The Ridge regression precision is: 0.7710280373831776
C = 0.05, The Ridge regression precision is: 0.7757009345794392
C = 0.06, The Ridge regression precision is: 0.7710280373831776
C = 0.03, The Ridge regression precision is: 0.7710280373831776
```

Neural Network Extension

In [1]:
```
from sklearn.datasets import load_digits
from sklearn.datasets import load_iris
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
import numpy as np
import numpy.random as r
import matplotlib.pyplot as plt
```

Creating the neural network

The activation function and its derivative

We will use the sigmoid activation function: $f(z)=\frac{1}{1+e^{-z}}$

The derivative of the sigmoid function is: $f'(z) = f(z)(1-f(z))$

In [2]:
```
def f(z):
    return 1 / (1 + np.exp(-z))

def f_deriv(z):
    return f(z) * (1 - f(z))
```

Creating and initialing W and b

We want the weights in W to be different so that during back propagation the nodes on a level will have different gradients and thus have different update values.

We want the weights to be small values, since the sigmoid is almost "flat" for large inputs.

Next is the code that assigns each weight a number uniformly drawn from $[0.0, 1.0)$. The code assumes that the number of neurons in each level is in the python list *nn_structure*.

In the code, the weights, W^{ℓ} and b^{ℓ} are held in a python dictionary

In [3]:
```
def setup_and_init_weights(nn_structure):
    W = {} #creating a dictionary i.e. a set of key: value pairs
    b = {}
    for l in range(1, len(nn_structure)):
        a = np.sqrt(6 / (nn_structure[l] + nn_structure[l-1]))
        W[l] = 2*a* r.random_sample((nn_structure[l], nn_structure[l-1])) - a #Return "continuous uniform" random floats in the half-open interval [0.0, 1.0).
        b[l] = 2*a* r.random_sample((nn_structure[l],)) - a
    return W, b
```

Initializing $\triangledown W$ and $\triangledown b$

Creating $\triangledown W^{\ell}$ and $\triangledown b^{\ell}$ to have the same size as W^{ℓ} and b^{ℓ}, and setting $\triangledown W^{\ell}$, and $\triangledown b^{\ell}$ to zero

In [4]:
```
def init_tri_values(nn_structure):
    tri_W = {}
    tri_b = {}
    for l in range(1, len(nn_structure)):
        tri_W[l] = np.zeros((nn_structure[l], nn_structure[l-1]))
        tri_b[l] = np.zeros((nn_structure[l],))
    return tri_W, tri_b
```

Feed forward

Perform a forward pass throught the network. The function returns the values of a and z

In [5]:
```
def feed_forward(x, W, b):
    a = {1: x} # create a dictionary for holding the a values for all levels
    z = { } # create a dictionary for holding the z values for all the layers
    for l in range(1, len(W) + 1): # for each layer
        node_in = a[l]
        z[l+1] = W[l].dot(node_in) + b[l]  # z^(l+1) = W^(l)*a^(l) + b^(l)
        a[l+1] = f(z[l+1]) # a^(l+1) = f(z^(l+1))
    return a, z
```

Compute δ

The code below compute $\delta^{(s_l)}$ in a function called "calculate_out_layer_delta", and computes $\delta^{(\ell)}$ for the hidden layers in the function called "calculate_hidden_delta".

If we wanted to have a different cost function, we would change the "calculate_out_layer_delta" function.

In [6]:
```
def calculate_out_layer_delta(y, a_out, z_out):
    # delta^(nl) = -(y_i - a_i^(nl)) * f'(z_i^(nl))
    return -(y-a_out) * f_deriv(z_out)


def calculate_hidden_delta(delta_plus_1, w_l, z_l):
    # delta^(l) = (transpose(W^(l)) * delta^(l+1)) * f'(z^(l))
    return np.dot(np.transpose(w_l), delta_plus_1) * f_deriv(z_l)
```

Initialize hw and hb for Ada Grad

In [7]:
```
def init_h_w_b_values(nn_structure):
    h_w = {}
    h_b = {}
    for l in range(1, len(nn_structure)):
        h_w[l] = np.zeros((nn_structure[l], nn_structure[l-1]))
        h_b[l] = np.zeros((nn_structure[l],))
    return h_w, h_b
```

The Back Propagation Algorithm

In [17]:
```
def train_nn(nn_structure, X, y, iter_num, alpha, lamb):
    W, b = setup_and_init_weights(nn_structure)
    cnt = 0
    N = len(y)
    avg_cost_func = []
    print('Starting gradient descent for {} iterations'.format(iter_num))
    h_w, h_b = init_h_w_b_values(nn_structure)
    while cnt < iter_num:
        if cnt%1000 == 0:
            print('Iteration {} of {}'.format(cnt, iter_num))
        tri_W, tri_b = init_tri_values(nn_structure)
        avg_cost = 0
        for i in range(N):
            delta = {}
            # perform the feed forward pass and return the stored a and z values, to be used in the
            # gradient descent step
            a, z = feed_forward(X[i, :], W, b)
            # loop from nl-1 to 1 backpropagating the errors
            for l in range(len(nn_structure), 0, -1):
                if l == len(nn_structure):
                    delta[l] = calculate_out_layer_delta(y[i,:], a[l], z[l])
                    avg_cost += np.linalg.norm(y[i,:]-a[l])
                else:
                    if l > 1:
                        delta[l] = calculate_hidden_delta(delta[l+1], W[l], z[l])
                    # triW^(l) = triW^(l) + delta^(l+1) * transpose(a^(l))
                    tri_W[l] += np.dot(delta[l+1][:,np.newaxis], np.transpose(a[l][:,np.newaxis]))# np.newaxis increase the number of dimensions
                    # trib^(l) = trib^(l) + delta^(l+1)
                    tri_b[l] += delta[l+1]
            # perform the gradient descent step for the weights in each layer
            for l in range(len(nn_structure) - 1, 0, -1):
                h_w[l] += tri_W[l] * tri_W[l]
                h_b[l] += tri_b[l] * tri_b[l]

                W[l] += -alpha * (tri_W[l]/(np.sqrt(h_w[l]) + 1e-7) + lamb/2*W[l])
                b[l] += -alpha * (tri_b[l]/(np.sqrt(h_b[l]) + 1e-7))
            # complete the average cost calculation
            # Remain the
            avg_cost = 1.0/N * avg_cost
            avg_cost_func.append(avg_cost)
            cnt += 1
    return W, b, avg_cost_func


def predict_y(W, b, X, n_layers):
    N = X.shape[0]
    y = np.zeros((N,))
    for i in range(N):
        a, z = feed_forward(X[i, :], W, b)
        y[i] = np.argmax(a[n_layers])
    return y
```

One hot encoding

Our target is an integer in the range [0,...,9], so we will have 10 output neuron's in our network.

- If $y=0$, we want the output neurons to have the values $(1,0,0,0,0,0,0,0,0,0)$
- If $y=1$ we want the output neurons to have the values $(0,1,0,0,0,0,0,0,0,0)$
- etc

Thus we need to change our target so it is the same as our hoped for output of the neural network.

- If $y=0$ we change it into the vector $(1,0,0,0,0,0,0,0,0,0)$
- If $y=1$ we change it into the vector $(0,1,0,0,0,0,0,0,0,0)$
- etc

See page 29 from the website listed above

The code to covert the target vector.

In [18]:
```
def convert_y_to_vect(y, num_class):
    y_vect = np.zeros((len(y), num_class))
    for i in range(len(y)):
        y_vect[i, y[i]] = 1
    return y_vect
```

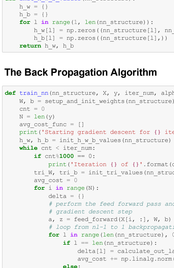
Test on Handwritten Digits data

In [19]:
```
digits=load_digits()
X = digits.data
y = digits.target

X_scale = StandardScaler()
X = X_scale.fit_transform(digits.data)

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.4)

y_v_train = convert_y_to_vect(y_train, 10)
y_v_test = convert_y_to_vect(y_test, 10)


nn_structure = [64, 30, 10]

# train the NN
alpha = 0.25
lamb = 0.01
W, b, avg_cost_func = train_nn(nn_structure, X_train, y_v_train, 3000, alpha, lamb)
```

```
Starting gradient descent for 3000 iterations
Iteration 0 of 3000
Iteration 1000 of 3000
Iteration 2000 of 3000
```

Plotting the learning curve for handwritten digits data for Adagrad

In [20]:
```
# plot the avg_cost_func
plt.plot(avg_cost_func)
plt.ylabel('Average J')
plt.xlabel('Iteration number')
plt.show()
```



Prediction accuracy for handwritten digits data for Adagrad

In [21]:
```
# get the prediction accuracy and print
y_pred = predict_y(W, b, X_test, 3)
print('Prediction accuracy is {}%'.format(accuracy_score(y_test, y_pred) * 100))
```

```
Prediction accuracy is 97.35744089012516%
```

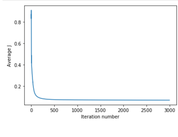
Test on Iris data

In [23]:
```
iris = load_iris()
X = iris.data
y = iris.target

X_scale = StandardScaler()
X = X_scale.fit_transform(iris.data)

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.4)

y_v_train = convert_y_to_vect(y_train, 3)
y_v_test = convert_y_to_vect(y_test, 3)

nn_structure = [4, 30, 3]

alpha = 0.25
lamb = 0.01
W, b, avg_cost_func = train_nn(nn_structure, X_train, y_v_train, 3000, alpha, lamb)
```

```
Starting gradient descent for 3000 iterations
Iteration 0 of 3000
Iteration 1000 of 3000
Iteration 2000 of 3000
```

Plotting the learning curve for iris data for Adagrad

In [56]:
```
# plot the avg_cost_func
plt.plot(avg_cost_func)
plt.ylabel('Average J')
plt.xlabel('Iteration number')
plt.show()
```



Prediction accuracy for iris data for Adagrad

In [57]:
```
# get the prediction accuracy and print
y_pred = predict_y(W, b, X_test, 3)
print('Prediction accuracy is {}%'.format(accuracy_score(y_test, y_pred) * 100))
```

```
Prediction accuracy is 96.66666666666667%
```