Homework 4

2021年11月6日 星期六　　下午7:47

Problem 1

1. $\epsilon_1 = \frac{5}{16} = 0.3125$

$\alpha_1 = \frac{1}{2}\log\left(\frac{1-\epsilon_1}{\epsilon_1}\right) = 0.39$

2. $\frac{1}{16} e^{-0.39\cdot(+1)(+1)} = 0.04231605466$

$\frac{1}{16} e^{-0.39(-1)(+1)} = 0.09231129962$

$w_1' = \frac{1}{11\times 0.04231605466 + 5\times 0.09231129962} \cdot 0.04231605466$

$= 0.04564675705$ (the instaces predicted correct)

$w_2' = \frac{1}{11\times 0.04231605466 + 5\times 0.09231129962} \cdot 0.09231129962$

$= 0.09957713345$ (the instances predicted incorrect)

3. Yes. In this case, $\alpha_i = +\infty$, and the weights of all examples are 0.

Problem 2

1. Boosted Decision Trees: Boosted decision trees use an efficient implementation of the MART gradient boosting algorithm. Gradient boosting is a machine learning technique for regression problems. It builds each regression tree in a stepwise fashion, using a predefined loss function to measure the error in each step and correct for it in the next. Gradient boosting works by building simpler (weak) prediction models sequentially where each model tries to predict the error left over by the previous model. Because of this, the algorithm tends to overfit rather quick.

   Training methodology

   Load the prepared data -> load_svmlight_file('a9a.txt')
   Train the XGBoost Model -> split the training data and use the K-fold CV
   Tune the hyperparameters -> use grid search
   Decide the final classifiers
   Make Predictions with XGBoost Model -> model.score(X_test, y_test)

   List of hyperparameter: eta, gamma, max_depth, min_child_weight, max_delta_step, subsample, sampling_method, colsample_bytree, colsample_bylevel, colsample_bynode, lambda, alpha, tree_method, sketch_eps, scale_pos_weight, updater, refresh_leaf, process_type, grow_policy, max_leaves, max_bin, predictor, num_parallel_tree, monotone_constraints, interaction_constraints

   Hyperparameter I tuned:

   tree method [default = auto]: For training boosted tree models
   max_depth [default = 6]: Maximum depth of a tree.
   min_child_weight [default = 1]: Minimum sum of instance weight (hessian) needed in a child.
   eta [default = .3]: Step size shrinkage used in update to prevents overfitting.
   colsample_bytree [default = 1]: is the subsample ratio of columns when constructing each tree.

   final hyperparameter settings:

   tree method [default = auto]
   max_depth = 7
   min_child_weight [default = 1]:
   eta [default = .3]:
   colsample_bytree = 5:

   Training error rates: 0.1542
   Cross-validation error rates: 0.1535626536
   test error rates: 0.1518

2. Random Forest: As in bagging, we build several decision trees on bootstrapped training samples. But when building these decision trees, each time a split in a tree is considered, a random sample of p < d features (predictors) is chosen as split candidates from the full set of all d features. The split is allowed to use only one of those p features. A fresh sample of features is taken at each split, and typically we choose $p < \sqrt{d}$

Training methodology

Load the prepared data -> load_svmlight_file('a9a.txt')
Train the RandomForest Model -> split the training data and use the K-fold CV
clf.fit(X_train, y_train)
Tune the hyperparameters -> use grid search
Decide the final classifiers
Make Predictions with RandomForest Model -> clfl.score(X_test, y_test)

List of hyperparameter: {'bootstrap': True, 'ccp_alpha': 0.0, 'class_weight': None, 'criterion': 'gini', 'max_depth': None, 'max_features': 'auto', 'max_leaf_nodes': None, 'max_samples': None, 'min_impurity_decrease': 0.0, 'min_impurity_split': None, 'min_samples_leaf': 1, 'min_samples_split': 2, 'min_weight_fraction_leaf': 0.0, 'n_estimators': 100, 'n_jobs': None, 'oob_score': False, 'random_state': None, 'verbose': 0, 'warm_start': False}

Hyperparameter I tuned:

n_estimators [default = 100]: number of trees in the foreset
max_features [default = auto] = max number of features considered for splitting a node
max_depth [default = None] = max number of levels in each decision tree
min_samples_split [default = 2] = min number of data points placed in a node before the node is split
min_samples_leaf [default = 1] = min number of data points allowed in a leaf node
bootstrap [default = True] = method for sampling data points (with or without replacement)

final hyperparameter settings:

{'n_estimators': 200,
'min_samples_split': 5,
'min_samples_leaf': 2,
'max_features': 'auto',
'bootstrap': False}

Training error rates: 0.1533
Cross-validation error rates: 0.1565520066
test error rates: 0.155

3. Support Vector Machines with Gaussian Kernel: In SVM, kernels are used for solving nonlinear problems in higher dimensional where linear separation is not possible. Generally, SVM is a simple dot product operation. Therefore, we should choose a kernel such that, K(x,y)=K(x).K(y). Gaussian is one such kernel giving good linear separation in higher dimension for many nonlinear problems. How it works? $\Phi: R \rightarrow R^{\infty} \quad \Phi(x) =$ $\exp(-x^2)(1, \sqrt{\frac{2^1}{1!}}x, \sqrt{\frac{2^2}{2!}}x^2, \sqrt{\frac{2^3}{3!}}x^3, \ldots)$

Training methodology

Load the prepared data -> load_svmlight_file('a9a.txt')
Train the SVM Gaussian Model -> split the training data and use the K-fold CV
clf.fit(X_train, y_train)
Tune the hyperparameters -> use grid search
Decide the final classifiers
Make Predictions with SVM Gaussian Model -> clfl.score(X_test, y_test)

List of hyperparameter: {'C': 1.0, 'break_ties': False, 'cache_size': 200, 'class_weight': None, 'coef0': 0.0, 'decision_function_shape': 'ovr', 'degree': 3, 'gamma': 'scale', 'kernel': 'rbf', 'max_iter': -1, 'probability': False, 'random_state': None, 'shrinking': True, 'tol': 0.001, 'verbose': False}

Hyperparameter I tuned:

Kernel[rbf]: The function of kernel is to take data as input and transform it into the required form.

final hyperparameter settings:

kernel = linear

Training error rates: 0.1569831716
Cross-validation error rates: 0.152047502
test error rates: 0.1502

Problem 3

jupyter Untitled Last Checkpoint: 9 minutes ago (autosaved)

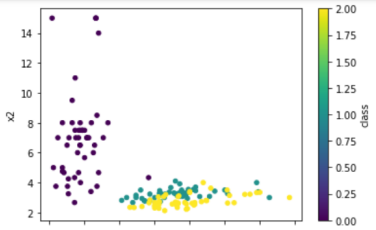
Logout

File Edit View Insert Cell Kernel Widgets Help

Trusted | Python 3

Run Code

```
    elif data_map[i][1] == centers[1]:
        label.append(1)
    elif data_map[i][1] == centers[2]:
        label.append(2)
    elif data_map[i][1] == centers[3]:
        label.append(3)
    elif data_map[i][1] == centers[4]:
        label.append(4)
plt.scatter(x1, x2, c=label)
plt.show()
```



jupyter Problem3.py 10 minutes ago

Logout

File Edit View Language Python

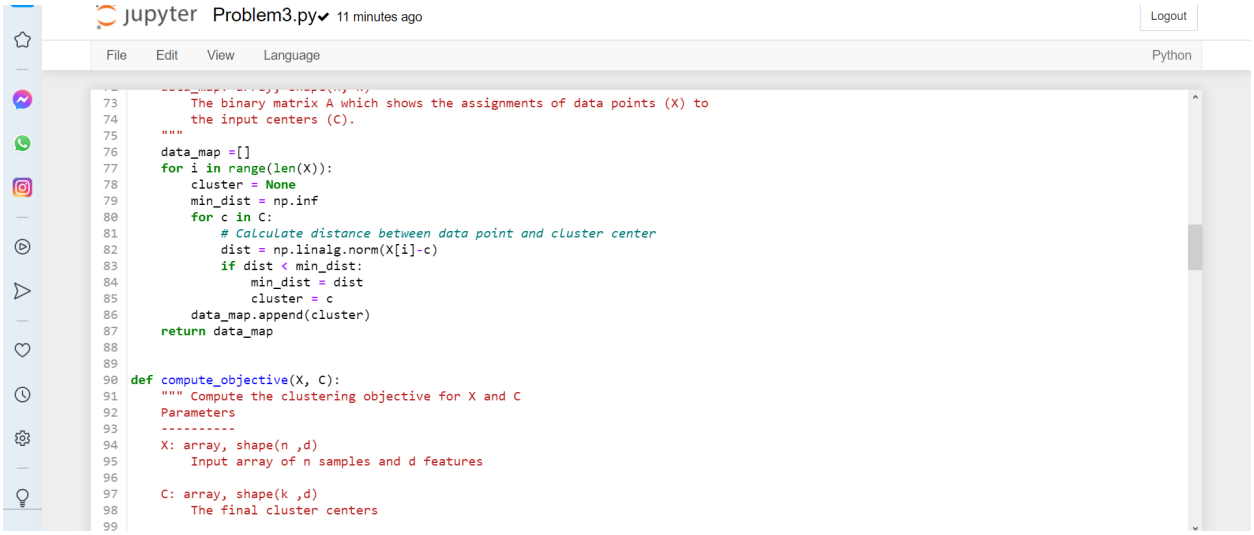
```
import numpy as np
import scipy as sp
import pandas as pd
import matplotlib
import matplotlib.pyplot as plt

data = np.loadtxt('iris.data', delimiter = ",", dtype='str')
#print(data)

temp = pd.DataFrame(data, columns=list('abcde'))
temp['e'] = [0 if item == 'Iris-setosa' else 1 if item == 'Iris-versicolor' else 2 for item in temp['e']]
temp = temp.astype(float)
x1 = temp['a']/temp['b']
x2 = temp['c']/temp['d']

new_data = pd.DataFrame()
new_data['x1'] = x1
new_data['x2'] = x2
new_data['class'] = temp['e']

new_data.plot.scatter('x1', 'x2', c='class', colormap='viridis')
plt.show()


def k_init(X, k):
    """ k-means++: initialization algorithm
```

jupyter Problem3.py 10 minutes ago

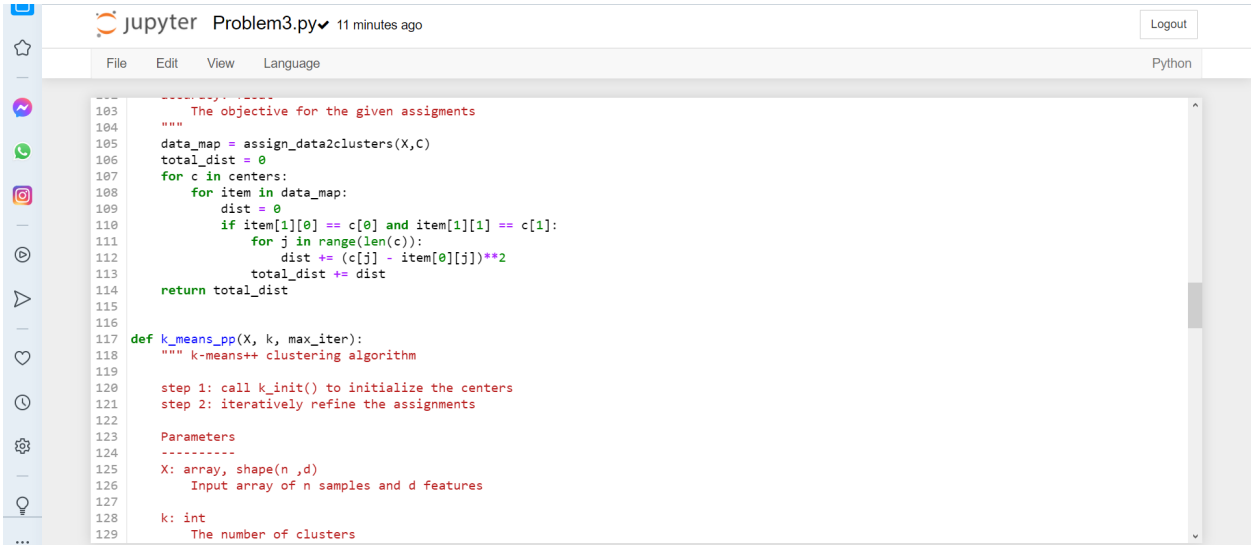
Logout

File Edit View Language Python

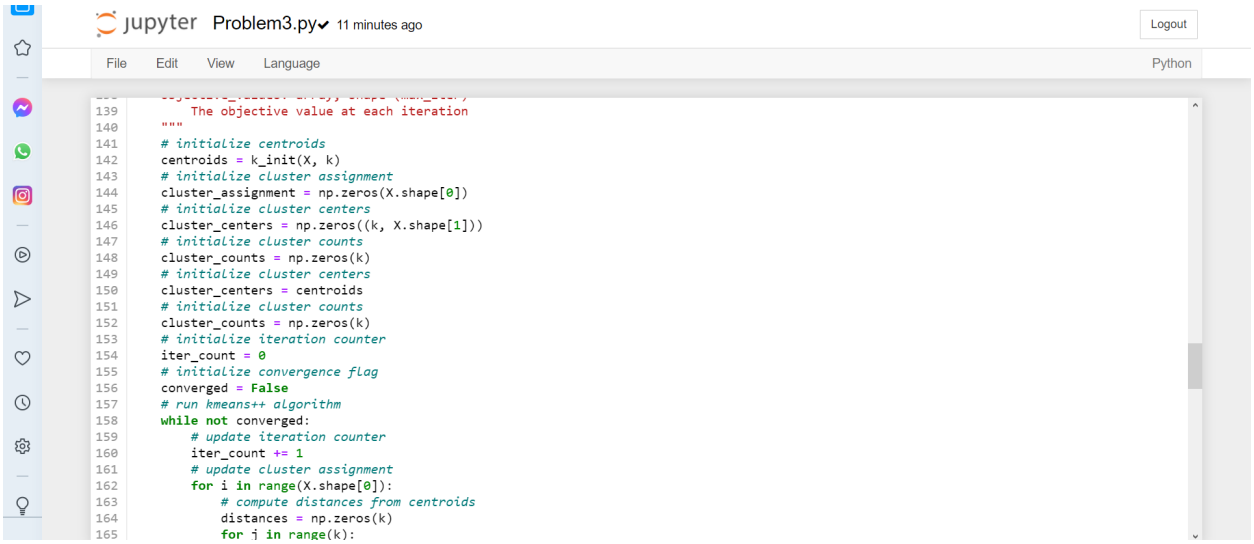
```
    -------
    init_centers: array (k, d)
        The initialize centers for kmeans++
    """
    # initialize k centroids
    centroids = np.zeros((k, X.shape[1]))
    #print(centroids[0])
    centroids[0] = X[np.random.randint(X.shape[0])]
    #print(X[0])
    #centroids[0].append(X[0])
    for i in range(1, k):
        # compute distances from centroids
        distances = np.zeros(X.shape[0])
        for j in range(X.shape[0]):
            distances[j] = np.linalg.norm(X[j] - centroids[i-1])
        # compute probabilities
        probabilities = distances / np.sum(distances)
        # choose centroid
        centroids[i] = X[np.random.choice(X.shape[0], 1, p=probabilities)]
        print(X.shape[0])
    return centroids


def assign_data2clusters(X, C):
    """ Assignments of data to the clusters
    Parameters
    ----------
```


jupyter Problem3.py 11 minutes ago
Logout
File Edit View Language
Python
The binary matrix A which shows the assignments of data points (X) to
the input centers (C).
"""
data_map =[]
for i in range(len(X)):
cluster = None
min_dist = np.inf
for c in C:
# Calculate distance between data point and cluster center
dist = np.linalg.norm(X[i]-c)
if dist < min_dist:
min_dist = dist
cluster = c
data_map.append(cluster)
return data_map
def compute_objective(X, C):
""" Compute the clustering objective for X and C
Parameters

X: array, shape(n ,d)
Input array of n samples and d features
C: array, shape(k ,d)
The final cluster centers


jupyter Problem3.py 11 minutes ago
Logout
File Edit View Language
Python
The objective for the given assigments
"""
data_map = assign_data2clusters(X,C)
total_dist = 0
for c in centers:
for item in data_map:
dist = 0
if item[1][0] == c[0] and item[1][1] == c[1]:
for j in range(len(c)):
dist += (c[j] - item[0][j])**2
total_dist += dist
return total_dist
def k_means_pp(X, k, max_iter):
""" k-means++ clustering algorithm
step 1: call k_init() to initialize the centers
step 2: iteratively refine the assignments
Parameters

X: array, shape(n ,d)
Input array of n samples and d features
k: int
The number of clusters




```
        The objective value at each iteration
    """
    # initialize centroids
    centroids = k_init(X, k)
    # initialize cluster assignment
    cluster_assignment = np.zeros(X.shape[0])
    # initialize cluster centers
    cluster_centers = np.zeros((k, X.shape[1]))
    # initialize cluster counts
    cluster_counts = np.zeros(k)
    # initialize cluster centers
    cluster_centers = centroids
    # initialize cluster counts
    cluster_counts = np.zeros(k)
    # initialize iteration counter
    iter_count = 0
    # initialize convergence flag
    converged = False
    # run kmeans++ algorithm
    while not converged:
        # update iteration counter
        iter_count += 1
        # update cluster assignment
        for i in range(X.shape[0]):
            # compute distances from centroids
            distances = np.zeros(k)
            for j in range(k):
```


```
            # compute distances from centroids
            distances = np.zeros(k)
            for j in range(k):
                distances[j] = np.linalg.norm(x[i] - centroids[j])
            # compute probabilities
            probabilities = distances / np.sum(distances)
            # choose centroid
            cluster_assignment[i] = np.random.choice(k, 1, p=probabilities)
        # update cluster centers
        for i in range(k):
            # compute cluster counts
            cluster_counts[i] = np.sum(cluster_assignment == i)
            # compute cluster centers
            cluster_centers[i] = np.sum(x[cluster_assignment == i], axis=0) / cluster_counts[i]
        # check for convergence
        if np.array_equal(centroids, cluster_centers):
            converged = True
        else:
            centroids = cluster_centers
    return centroids


cluster_data = pd.DataFrame()
cluster_data['x1'] = x1
cluster_data['x2'] = x2

#run with k=1,2,3,4,5
```

jupyter Problem3.py 12 minutes ago

Logout

File Edit View Language Python

```
cluster_data['x2'] = x2

#run with k=1,2,3,4,5
acc = []
centers = k_means_pp(cluster_data, 1, 50)
acc.append(compute_objective(cluster_data, centers))

centers = k_means_pp(cluster_data, 2, 50)
acc.append(compute_objective(cluster_data, centers))

centers = k_means_pp(cluster_data, 3, 50)
acc.append(compute_objective(cluster_data, centers))

centers = k_means_pp(cluster_data, 4, 50)
acc.append(compute_objective(cluster_data, centers))

centers = k_means_pp(cluster_data, 5, 50)
acc.append(compute_objective(cluster_data, centers))

plt.plot([1,2,3,4,5], acc)
plt.show()
plt.clf()

#my best was k=5 so now ill change the number fo iterations with k=5
acc = []
centers = k_means_pp(new_set, 5, 1)
acc.append(compute_objective(new_set, centers))
```

jupyter Problem3.py 12 minutes ago

Logout

File Edit View Language Python

```

centers = k_means_pp(new_set, 5, 50)
acc.append(compute_objective(new_set, centers))

centers = k_means_pp(new_set, 5, 100)
acc.append(compute_objective(new_set, centers))

centers = k_means_pp(new_set, 5, 200)
acc.append(compute_objective(new_set, centers))

plt.plot([1,20,50,100,200], acc)
plt.show()
plt.clf()

#plot with data colored by cluster
centers = k_means_pp(new_set, 5, 200)
data_map = assign_data2clusters(new_set, centers)
label = []
for i in range(len(data_map)):
    if data_map[i][1] == centers[0]:
        label.append(0)
    elif data_map[i][1] == centers[1]:
        label.append(1)
    elif data_map[i][1] == centers[2]:
        label.append(2)
    elif data_map[i][1] == centers[3]:
        label.append(3)
```

jupyter Problem3.py 13 minutes ago

Logout

File Edit View Language Python

```
acc.append(compute_objective(new_set, centers))

centers = k_means_pp(new_set, 5, 200)
acc.append(compute_objective(new_set, centers))

plt.plot([1,20,50,100,200], acc)
plt.show()
plt.clf()

#plot with data colored by cluster
centers = k_means_pp(new_set, 5, 200)
data_map = assign_data2clusters(new_set, centers)
label = []
for i in range(len(data_map)):
    if data_map[i][1] == centers[0]:
        label.append(0)
    elif data_map[i][1] == centers[1]:
        label.append(1)
    elif data_map[i][1] == centers[2]:
        label.append(2)
    elif data_map[i][1] == centers[3]:
        label.append(3)
    elif data_map[i][1] == centers[4]:
        label.append(4)
plt.scatter(x1, x2, c=label)
plt.show()
```

My final classifier is k = 5

All the code is in Jupyter notebook and problem3.