

KNN compute_distances

```
43     for i in np.arange(num_test):
44
45         for j in np.arange(num_train):
46             # ===== #
47             # YOUR CODE HERE:
48             #     Compute the distance between the ith test point and the jth
49
50             #     training point using norm(), and store the result in dists[i,
51             #     j].
52             # ===== #
53
54             dists[i,j] = norm(X[i,:]-self.X_train[j,:])
55
56             # ===== #
57             # END YOUR CODE HERE
58             # ===== #
59
60     return dists
```

compute_L2_distances_vectorized

```
76
77     # ===== #
78     # YOUR CODE HERE:
79     #     Compute the L2 distance between the ith test point and the jth
80     #     training point and store the result in dists[i, j]. You may
81     #     NOT use a for loop (or list comprehension). You may only use
82     #     numpy operations.
83     #
84     #     HINT: use broadcasting. If you have a shape (N,1) array and
85     #     a shape (M,) array, adding them together produces a shape (N, M)
86     #     array.
87     # ===== #
88
89
90     sum_test = np.sum(X**2,axis=1)
91     sum_train = np.sum(self.X_train**2,axis=1)
92     dists = np.sqrt(sum_test.reshape(-1,1) + sum_train -
93                     2*X.dot(self.X_train.T))
94
95     # ===== #
96     # END YOUR CODE HERE
97     # ===== #
98
99     return dists
100
```

predict_labels

```
120     closest_y = []
121     # ===== #
122     # YOUR CODE HERE:
123     #   Use the distances to calculate and then store the labels of
124     #   the k-nearest neighbors to the ith test point. The function
125     #   numpy.argsort may be useful.
126     #
127     #   After doing this, find the most common label of the k-nearest
128     #   neighbors. Store the predicted label of the ith training example
129     #   as y_pred[i]. Break ties by choosing the smaller label.
130     # ===== #
131
132     idx = np.argsort(dists[i,:])
133     closest_y = list(self.y_train[idx[:k]])
134     y_pred[i] = max(set(closest_y), key = closest_y.count)
135
136     # ===== #
137     # END YOUR CODE HERE
138     # ===== #
139
```

SVM loss

```
40     for i in np.arange(num_train):
41         # ===== #
42         # YOUR CODE HERE:
43         #     Calculate the normalized SVM loss, and store it as 'loss'.
44         #     (That is, calculate the sum of the losses of all the training
45         #     set margins, and then normalize the loss by the number of
46         #     training examples.)
47         # ===== #
48         score = X[i].dot(self.W.T)
49         for j in range(num_classes):
50             if(j != y[i]):
51                 loss += np.maximum(0, score[j] - score[y[i]] + 1)
52
53     loss /= num_train
54
55     # ===== #
56     # END YOUR CODE HERE
57     # ===== #
58
59     return loss
```

loss_and_grad

```
74
75     for i in np.arange(num_train):
76         # ===== #
77         # YOUR CODE HERE:
78         #     Calculate the SVM loss and the gradient.  Store the gradient in
79         #     the variable grad.
80         # ===== #
81         score = X[i].dot(self.W.T)
82         for j in range(num_classes):
83             if(j != y[i]):
84                 a = score[j] - score[y[i]] + 1
85                 loss += np.maximum(0, a)
86                 if (a > 0):
87                     grad[j,:] += X[i,:]
88                     grad[y[i],:] -= X[i,:]
89
90
91     # ===== #
92     # END YOUR CODE HERE
93     # ===== #
94
```

fast_loss_and_grad

```

129 # ===== #
130 # YOUR CODE HERE:
131 #   Calculate the SVM loss WITHOUT any for loops.
132 # ===== #
133
134 score = X.dot(self.W.T)
135 margin = np.maximum(0, score - score[np.arange(X.shape[0]),
136         y][:,np.newaxis] + 1)
137 margin[np.arange(X.shape[0]), y] -= 1
138
139 loss = np.sum(margin)
140 loss /= X.shape[0]
141
142 # ===== #
143 # END YOUR CODE HERE
144 # ===== #
145
146
147 # ===== #
148 # YOUR CODE HERE:
149 #   Calculate the SVM grad WITHOUT any for loops.
150 # ===== #
151
152 X_masked = np.zeros(margin.shape)
153 X_masked[margin > 0] = 1
154
155 count = np.sum(X_masked,axis=1)
156 X_masked[np.arange(X.shape[0]),y] = -count
157
158 grad = (X.T.dot(X_masked)).T
159 grad /= X.shape[0]
160
161 # ===== #
162 # END YOUR CODE HERE
163 # ===== #
164

```

train

```
197     # ===== #
198     # YOUR CODE HERE:
199     #   Sample batch_size elements from the training data for use in
200     #   gradient descent. After sampling,
201     #       - X_batch should have shape: (dim, batch_size)
202     #       - y_batch should have shape: (batch_size,)
203     #   The indices should be randomly generated to reduce correlations
204     #   in the dataset. Use np.random.choice. It's okay to sample with
205     #   replacement.
206     # ===== #
207
208     idx = np.random.choice(num_train, batch_size)
209     X_batch = X[idx,:]
210     y_batch = y[idx]
211
212     # ===== #
213     # END YOUR CODE HERE
214     # ===== #
215
216     # evaluate loss and gradient
217     loss, grad = self.fast_loss_and_grad(X_batch, y_batch)
218     loss_history.append(loss)
219
220     # ===== #
221     # YOUR CODE HERE:
222     #   Update the parameters, self.W, with a gradient step
223     # ===== #
224
225     self.W -= learning_rate * grad
226
227     # ===== #
228     # END YOUR CODE HERE
229     # ===== #
230
```

Predict

```
240
249     # ===== #
250     # YOUR CODE HERE:
251     #   Predict the labels given the training data with the parameter self.W.
252     # ===== #
253
254     pred = X.dot(self.W.T)
255
256     y_pred = np.argmax(pred, axis = 1)
257
258     # ===== #
259     # END YOUR CODE HERE
260     # ===== #
261
```

Softmax loss

```
32 # Initialize the loss to zero.
33 loss = 0.0
34
35 # ===== #
36 # YOUR CODE HERE:
37 # Calculate the normalized softmax loss. Store it as the variable
38 # loss.
39 # (That is, calculate the sum of the losses of all the training
40 # set margins, and then normalize the loss by the number of
41 # training examples.)
42 # ===== #
43 a = X.dot(self.W.T)
44
45 for i in range(X.shape[0]):
46     a[i,:] -= np.max(a[i,:])
47     loss += -np.log(np.exp(a[i,y[i]])/np.sum(np.exp(a[i,:])))
48
49 loss /= X.shape[0]
50
51 # ===== #
52 # END YOUR CODE HERE
53 # ===== #
54 return loss
```

loss_and_grad

```
64 # Initialize the loss and gradient to zero.
65 loss = 0.0
66 grad = np.zeros_like(self.W)
67
68 # ===== #
69 # YOUR CODE HERE:
70 # Calculate the softmax loss and the gradient. Store the gradient
71 # as the variable grad.
72 # ===== #
73 a = X.dot(self.W.T)
74
75 for i in range(X.shape[0]):
76     a[i,:] -= np.max(a[i,:])
77     loss += -np.log(np.exp(a[i,y[i]])/np.sum(np.exp(a[i,:])))
78
79     for j in range(self.W.shape[0]):
80         grad[j] += (np.exp(a[i,j])/np.sum(np.exp(a[i,:]))) - (y[i] == j))
81         * X[i]
82
83 loss /= X.shape[0]
84 grad /= X.shape[0]
85
86 # ===== #
87 # END YOUR CODE HERE
88 # ===== #
89 return loss, grad
```

fast_loss_and_grad

```
120     # ===== #
121     # YOUR CODE HERE:
122     #   Calculate the softmax loss and gradient WITHOUT any for loops.
123     # ===== #
124
125     a = X.dot(self.W.T)
126     a -= np.max(a, axis=1, keepdims=True)
127     score = np.exp(a)/np.sum(np.exp(a), axis=1, keepdims=True)
128     loss = np.sum(-np.log(score[np.arange(X.shape[0]), y]))
129
130     score[range(y.shape[0]),y] -= 1
131     grad = score.T.dot(X)
132     grad /= y.shape[0]
133     loss /= y.shape[0]
134
135     # ===== #
136     # END YOUR CODE HERE
137     # ===== #
138
139     return loss, grad
```

train and predict are the same as SVM