KNN compute_distances

```
for i in np.arange(num_test):

  for j in np.arange(num_train):
    # ================================================================ #
    # YOUR CODE HERE:
    #   Compute the distance between the ith test point and the jth
    #   training point using norm(), and store the result in dists[i, j].
    # ================================================================ #

    dists[i,j] = norm(X[i,:]-self.X_train[j,:])

    # ================================================================ #
    # END YOUR CODE HERE
    # ================================================================ #

return dists
```

compute_L2_distances_vectorized

```
# ================================================================ #
# YOUR CODE HERE:
#   Compute the L2 distance between the ith test point and the jth
#   training point and store the result in dists[i, j].  You may
#   NOT use a for loop (or list comprehension).  You may only use
#   numpy operations.
#
#   HINT: use broadcasting.  If you have a shape (N,1) array and
#   a shape (M,) array, adding them together produces a shape (N, M)
#   array.
# ================================================================ #


sum_test = np.sum(X**2,axis=1)
sum_train = np.sum(self.X_train**2,axis=1)
dists = np.sqrt(sum_test.reshape(-1,1) + sum_train - 2*X.dot(self.X_train.T))


# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #

return dists
```

predict_labels

```
closest_y = []
# ================================================================ #
# YOUR CODE HERE:
#   Use the distances to calculate and then store the labels of
#   the k-nearest neighbors to the ith test point.  The function
#   numpy.argsort may be useful.
#
#   After doing this, find the most common label of the k-nearest
#   neighbors.  Store the predicted label of the ith training example
#   as y_pred[i].  Break ties by choosing the smaller label.
# ================================================================ #

idx = np.argsort(dists[i,:])
closest_y = list(self.y_train[idx[:k]])
y_pred[i] = max(set(closest_y), key = closest_y.count)

# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #
```

SVM loss

```python
        for i in np.arange(num_train):
        # ================================================================ #
        # YOUR CODE HERE:
          #   Calculate the normalized SVM loss, and store it as 'loss'.
        #   (That is, calculate the sum of the losses of all the training
        #   set margins, and then normalize the loss by the number of
          #   training examples.)
        # ================================================================ #
            score = X[i].dot(self.W.T)
            for j in range(num_classes):
                if(j != y[i]):
                    loss += np.maximum(0,score[j] - score[y[i]] + 1)

        loss /= num_train

        # ================================================================ #
        # END YOUR CODE HERE
        # ================================================================ #

        return loss
```

loss_and_grad

```python

        for i in np.arange(num_train):
        # ================================================================ #
        # YOUR CODE HERE:
        #   Calculate the SVM loss and the gradient.  Store the gradient in
        #   the variable grad.
        # ================================================================ #
            score = X[i].dot(self.W.T)
            for j in range(num_classes):
                if(j != y[i]):
                    a = score[j] - score[y[i]] + 1
                    loss += np.maximum(0,a)
                    if (a>0):
                        grad[j,:] += X[i,:]
                        grad[y[i],:] -= X[i,:]


        # ================================================================ #
        # END YOUR CODE HERE
        # ================================================================ #
```

fast_loss_and_grad

```python
        # ================================================================ #
        # YOUR CODE HERE:
        #   Calculate the SVM loss WITHOUT any for loops.
        # ================================================================ #

        score = X.dot(self.W.T)
        margin = np.maximum(0,score - score[np.arange(X.shape[0]),y][:,np.newaxis] + 1)
        margin[np.arange(X.shape[0]), y] -= 1

        loss = np.sum(margin)
        loss /= X.shape[0]

        # ================================================================ #
        # END YOUR CODE HERE
        # ================================================================ #



        # ================================================================ #
        # YOUR CODE HERE:
        #   Calculate the SVM grad WITHOUT any for loops.
        # ================================================================ #

        X_masked = np.zeros(margin.shape)
        X_masked[margin > 0] = 1

        count = np.sum(X_masked,axis=1)
        X_masked[np.arange(X.shape[0]),y] = -count

        grad = (X.T.dot(X_masked)).T
        grad /= X.shape[0]

        # ================================================================ #
        # END YOUR CODE HERE
        # ================================================================ #
```

train

```
        # ================================================================ #
        # YOUR CODE HERE:
        #   Sample batch_size elements from the training data for use in
        #   gradient descent.  After sampling,
        #     - X_batch should have shape: (dim, batch_size)
          #     - y_batch should have shape: (batch_size,)
          #   The indices should be randomly generated to reduce correlations
          #   in the dataset.  Use np.random.choice.  It's okay to sample with
          #   replacement.
        # ================================================================ #

        idx = np.random.choice(num_train, batch_size)
        X_batch = X[idx,:]
        y_batch = y[idx]

        # ================================================================ #
        # END YOUR CODE HERE
        # ================================================================ #

        # evaluate loss and gradient
        loss, grad = self.fast_loss_and_grad(X_batch, y_batch)
        loss_history.append(loss)

        # ================================================================ #
        # YOUR CODE HERE:
        #   Update the parameters, self.W, with a gradient step
        # ================================================================ #

        self.W -= learning_rate * grad

        # ================================================================ #
        # END YOUR CODE HERE
        # ================================================================ #
```

Predict

```
    # ================================================================ #
    # YOUR CODE HERE:
    #   Predict the labels given the training data with the parameter self.W.
    # ================================================================ #

    pred = X.dot(self.W.T)

    y_pred = np.argmax(pred, axis = 1)

    # ================================================================ #
    # END YOUR CODE HERE
    # ================================================================ #
```

Softmax `loss`

```
# Initialize the loss to zero.
loss = 0.0

# ================================================================ #
# YOUR CODE HERE:
#   Calculate the normalized softmax loss.  Store it as the variable
    loss.
#   (That is, calculate the sum of the losses of all the training
#   set margins, and then normalize the loss by the number of
#   training examples.)
# ================================================================ #
a = X.dot(self.W.T)

for i in range(X.shape[0]):
    a[i,:] -= np.max(a[i,:])
    loss += -np.log(np.exp(a[i,y[i]])/np.sum(np.exp(a[i,:])))

loss /= X.shape[0]

# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #

return loss
```

`loss_and_grad`

```
# Initialize the loss and gradient to zero.
loss = 0.0
grad = np.zeros_like(self.W)

# ================================================================ #
# YOUR CODE HERE:
#   Calculate the softmax loss and the gradient. Store the gradient
#   as the variable grad.
# ================================================================ #
a = X.dot(self.W.T)

for i in range(X.shape[0]):
    a[i,:] -= np.max(a[i,:])
    loss += -np.log(np.exp(a[i,y[i]])/np.sum(np.exp(a[i,:])))

    for j in range(self.W.shape[0]):
        grad[j] += (np.exp(a[i,j])/np.sum(np.exp(a[i,:])) - (y[i] == j))
            * X[i]

loss /= X.shape[0]
grad /= X.shape[0]

# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #

return loss, grad
```

fast_loss_and_grad

```python
    # ================================================================ #
    # YOUR CODE HERE:
    #   Calculate the softmax loss and gradient WITHOUT any for loops.
    # ================================================================ #

    a = X.dot(self.W.T)
    a -= np.max(a, axis=1, keepdims=True)
    score = np.exp(a)/np.sum(np.exp(a), axis=1, keepdims=True)
    loss = np.sum(-np.log(score[np.arange(X.shape[0]), y]))

    score[range(y.shape[0]),y] -= 1
    grad = score.T.dot(X)
    grad /= y.shape[0]
    loss /= y.shape[0]

    # ================================================================ #
    # END YOUR CODE HERE
    # ================================================================ #

    return loss, grad
```

train and predict are the same as SVM