Distances and Angles between Images

We are going to compute distances and angles between images.

Learning objectives

By the end of this notebook, you will learn to

1. Write programs to compute distance.
2. Write programs to compute angle.

"distance" and "angle" are useful beyond their usual interpretation. They are useful for describing **similarity** between objects. You will first use the functions you wrote to compare MNIST digits. Furthermore, we will use these concepts for implementing the K Nearest Neighbors algorithm, which is a useful algorithm for classifying object according to distance.

In [42]:
```
# PACKAGE: DO NOT EDIT THIS LINE
import matplotlib as mpl
import matplotlib.pyplot as plt
import numpy as np
import scipy

import sklearn
from ipywidgets import interact
from load_data import load_mnist
```

The next cell loads the MNIST digits dataset.

In [43]:
```
MNIST = load_mnist()
images = MNIST['data'].astype(np.double)
labels = MNIST['target'].astype(np.int)
```

In [44]:
```
# Plot figures so that they can be shown in the notebook
%matplotlib inline
%config InlineBackend.figure_format = 'svg'
```

For this assignment, you need to implement the two functions (`distance` and `angle`) in the cell below which compute the distance and angle between two vectors.

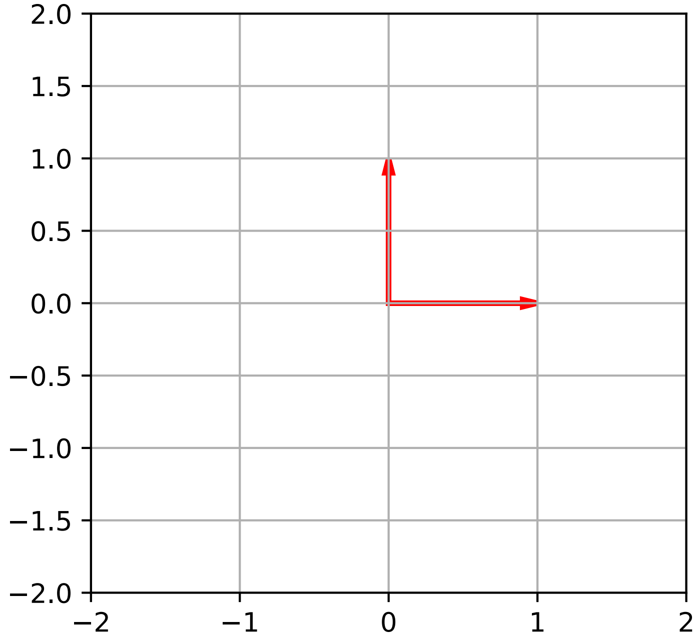
In [45]:
```
# GRADED FUNCTION: DO NOT EDIT THIS LINE

def distance(x0, x1):
    """Compute distance between two vectors x0, x1 using the dot product"""
    between = x0 - x1
    distance = np.linalg.norm(between) # <-- EDIT THIS to compute the distance
between x0 and x1
    return distance

def angle(x0, x1):
    """Compute the angle between two vectors x0, x1 using the dot product"""
    #arccos -> cos inverse, linalg.norm -> distance of vector
    angle = np.arccos(np.dot(x0, x1) / (np.linalg.norm(x0) * np.linalg.norm(x1
)))# <-- EDIT THIS to compute angle between x0 and x1
    return angle
```

We have created some helper functions for you to visualize vectors in the cells below. You do not need to modify them.

In [46]:
```
def plot_vector(v, w):
    fig = plt.figure(figsize=(4,4))
    ax = fig.gca()
    plt.xlim([-2, 2])
    plt.ylim([-2, 2])
    plt.grid()
    ax.arrow(0, 0, v[0], v[1], head_width=0.05, head_length=0.1,
             length_includes_head=True, linewidth=2, color='r');
    ax.arrow(0, 0, w[0], w[1], head_width=0.05, head_length=0.1,
             length_includes_head=True, linewidth=2, color='r');
```

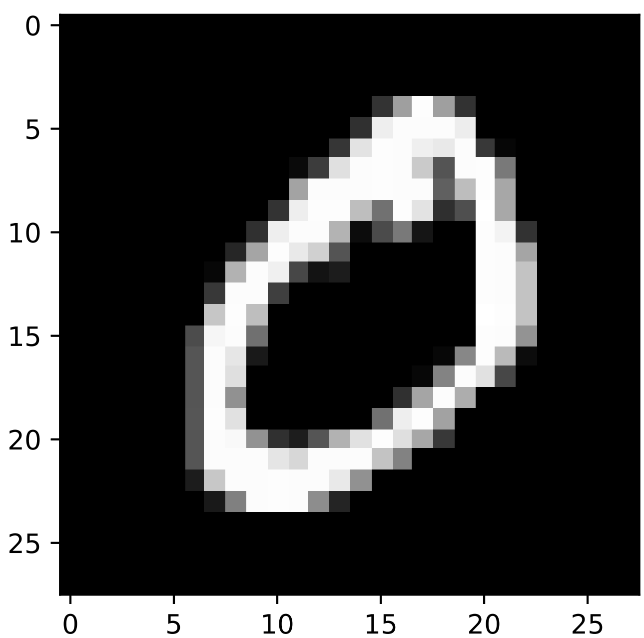
In [47]:
```
# Some sanity checks, you may want to have more interesting test cases to test
your implementation
a = np.array([1,0])
b = np.array([0,1])
np.testing.assert_almost_equal(distance(a, b), np.sqrt(2))
assert((angle(a,b) / (np.pi * 2) * 360.) == 90)
```

In [48]:
```
plot_vector(b, a)
```



The next cell shows some digits from the dataset.

In [49]:
```
plt.imshow(images[labels==0].reshape(-1, 28, 28)[0], cmap='gray');
```



But we have the following questions:

1. What does it mean for two digits in the MNIST dataset to be *different* by our distance function?
2. Furthermore, how are different classes of digits different for MNIST digits? Let's find out!

For the first question, we can see just how the distance between digits compare among all distances for the first 500 digits. The next cell computes pairwise distances between images.

In [50]:
```
#computes distance between images (the first 500 images in our dataset)
distances = []
for i in range(len(images[:500])):
    for j in range(len(images[:500])):
        distances.append(distance(images[i], images[j]))
```

In [51]:
```
@interact(first=(0, 499), second=(0, 499), continuous_update=False)
def show_img(first, second):
    plt.figure(figsize=(8,4))
    f = images[first].reshape(28, 28)
    s = images[second].reshape(28, 28)

    ax0 = plt.subplot2grid((2, 2), (0, 0))
    ax1 = plt.subplot2grid((2, 2), (1, 0))
    ax2 = plt.subplot2grid((2, 2), (0, 1), rowspan=2)

    #plt.imshow(np.hstack([f,s]), cmap='gray')
    ax0.imshow(f, cmap='gray')
    ax1.imshow(s, cmap='gray')
    ax2.hist(np.array(distances), bins=50)
    d = distance(f.ravel(), s.ravel())
    ax2.axvline(x=d, ymin=0, ymax=40000, color='C4', linewidth=4)
    ax2.text(0, 46000, "Distance is {:.2f}".format(d), size=12)
    ax2.set(xlabel='distance', ylabel='number of images')
    plt.show()
```

Next we will find the index of the most similar image to the image at index 0. We will do this by writing some code in another cell.

Write some code in this scratch cell below to find out the most similar image

In [58]:
```
#first 500 in the distances list is the similarity between all 500 images and
 the first image
print(sorted(distances[:500]))
```

```
[0.0, 1020.6473436011089, 1134.3138895385175, 1149.3206689170781, 1212.296168
4341001, 1267.5073964281235, 1269.680274714859, 1351.5568800461192, 1353.4906
722988526, 1362.8840743071291, 1364.1257273433414, 1374.4329739932755, 1388.0
05763676794, 1395.3641101877315, 1413.7068295795984, 1445.0885785999417, 144
5.8264072840834, 1448.0580098877253, 1453.8167009633642, 1456.200878999872, 1
476.4372658531754, 1480.2932141977819, 1487.1993813877143, 1492.344464257498
4, 1499.8733279847336, 1504.3403870135244, 1509.8466809580368, 1537.197124639
5174, 1539.9571422607839, 1571.8037409295093, 1575.6592271173358, 1593.070933
7628377, 1595.7079933371269, 1609.656485092394, 1610.2645745342595, 1629.3578
489699555, 1633.5981758070129, 1635.909227310611, 1641.3153261942082, 1655.68
29406622512, 1656.1391849720844, 1659.1283856290327, 1659.5095661068062, 166
9.3268104238905, 1677.4561097089843, 1680.6671294459234, 1684.5548373383397,
1690.2446568470496, 1690.7421447399956, 1695.2934849164023, 1696.538829499637
6, 1697.2695719890814, 1697.4586887462092, 1698.5405500016773, 1700.495516018
7867, 1703.2850612859845, 1706.8840030886693, 1709.5794804571094, 1710.044443
8668837, 1715.7773748362576, 1716.3481581543997, 1721.758693894124, 1725.5112
285928481, 1727.677342561394, 1728.512076903138, 1730.9818023306889, 1733.605
7798703832, 1740.403976092907, 1741.5541909455474, 1741.6414671223235, 1742.4
769725881602, 1745.2386656271399, 1746.1749625968184, 1753.7311652588032, 175
9.8391971995622, 1760.6674870627901, 1761.2958297798812, 1765.7244971965474,
1766.3736863982094, 1769.2781013735516, 1771.2128612902516, 1775.029577218362
9, 1775.4126280952269, 1777.8174821955149, 1792.8287704072579, 1794.791631360
0306, 1795.2359176442521, 1798.393171695222, 1799.7074762305124, 1799.9899999
72222, 1806.4268598534511, 1811.935705261089, 1813.7858197703499, 1815.001101
92804, 1820.795705179469, 1823.4823826952647, 1828.6062998907119, 1829.661717
3674483, 1833.5651065615314, 1838.3457237418645, 1839.5711456749914, 1841.242
243703962, 1844.1694607600464, 1844.6482049431538, 1844.8468771147377, 1846.4
238408339511, 1851.3851571188529, 1854.1092200838655, 1854.1138044898969, 186
5.2407351331356, 1865.7577549081768, 1866.7576168319229, 1867.9448599998877,
1868.4555119135161, 1868.8547295068174, 1870.784862029838, 1871.881139388930
3, 1873.8620546881245, 1876.0, 1877.6386766361627, 1878.3380952320592, 1878.8
366081168422, 1883.0467333552824, 1884.0204351333348, 1885.3031586458449, 188
5.5237999028282, 1890.0328039481219, 1893.0129423751966, 1895.1936576508481,
1896.2077945204212, 1901.3269050849724, 1908.6531900793293, 1909.072287787971
6, 1916.0430579712972, 1918.2846504103607, 1920.6756623646795, 1922.606824080
2642, 1931.2728963044037, 1936.1812931644599, 1943.9475301560997, 1949.030784
7748327, 1951.2311498128561, 1956.5960237105667, 1959.2034095519537, 1960.583
841614533, 1963.0433005922207, 1963.3644592892069, 1965.0106869938393, 1967.8
516204226376, 1968.9954291465483, 1973.3144706305684, 1974.272270989997, 197
5.4351419370873, 1977.4415288447849, 1980.8058965986547, 1981.7091108434659,
1984.288033527391, 1985.6757036334004, 1989.4403735724275, 1993.202699175374
8, 1994.8243030402452, 1995.5006890502443, 1995.5470427930281, 1996.681246468
7498, 2007.6242178256368, 2008.7351741829982, 2010.4228410958726, 2010.760055
302472, 2014.1089841416228, 2018.9569089012277, 2020.4209957333151, 2021.2706
399688291, 2023.5157523478783, 2023.647943689811, 2024.1247491199745, 2024.49
08001766764, 2025.1197495456904, 2026.9844597332265, 2027.495006159078, 2031.
498461727205, 2035.3557428616748, 2038.9507105371626, 2039.9348028797392, 204
0.5957463446796, 2041.5814948220902, 2042.0756597148893, 2043.2525541400896,
2043.5838617487661, 2045.5106941788399, 2053.6316612284686, 2055.769928761484
6, 2065.032929519527, 2065.0738485584479, 2067.8012477024963, 2068.6058106850
614, 2071.0362140725592, 2073.5949459814951, 2074.2820444674344, 2075.5192121
49095, 2075.812852836209, 2084.1902504330069, 2085.4985015578409, 2089.186683
8557057, 2092.5028076444723, 2099.005002376126, 2104.1345014043186, 2105.8076
835266793, 2107.9959677380789, 2108.3913299005949, 2115.6556903239243, 2116.9
033988351948, 2117.2271961223246, 2117.9848913530996, 2119.4055770427708, 212
0.6253794576733, 2121.8713438849209, 2125.5340505388285, 2125.6027380486694,
2129.3285326600026, 2130.648492830293, 2132.2985250663191, 2133.331666666015
```

```
7, 2133.6398477718772, 2136.0442411148697, 2136.4479867293749, 2140.210036421
6594, 2145.0967344154901, 2147.0589186140187, 2149.70579382389, 2150.30393200
5892, 2150.7189495608209, 2150.7277837978472, 2153.3956905315845, 2154.277837
234557, 2157.8718682998765, 2158.6155285274867, 2159.9030070815679, 2160.2518
371708425, 2161.0360940993096, 2162.9842810339605, 2163.0857588177128, 2164.4
119293701929, 2169.446934128604, 2170.134327639651, 2173.1120541748419, 2175.
2089554799099, 2176.3503853929406, 2178.0576209090523, 2178.2672012404723, 21
78.7363309955613, 2185.5260236382455, 2191.3370347803643, 2192.4442980381509,
2192.445438317679, 2192.525712506013, 2192.5927574449388, 2195.9924863259439,
2197.5397607324426, 2201.8169769533524, 2201.9940962681985, 2202.148496355320
7, 2202.9559686929742, 2204.477489111649, 2208.9004051790112, 2216.5691958520
042, 2217.7975561353655, 2220.1522470317209, 2223.3254372673382, 2224.5831070
11289, 2224.8604450616672, 2228.0305204372762, 2229.6537847836375, 2229.81030
58332113, 2232.0723554580395, 2236.3065532256528, 2240.7623702659771, 2242.65
44539897359, 2242.7050630878775, 2256.6951499925726, 2258.2136745666917, 226
1.6133621819622, 2263.7170759615701, 2263.7709689807402, 2263.7884176751149,
2268.9030389155018, 2271.202765056436, 2272.0930878817444, 2273.454420040129
2, 2277.7653522696319, 2278.7669911599123, 2280.6503458443603, 2281.161327043
7492, 2283.5301618327708, 2283.7342227150689, 2286.0126858790613, 2294.925924
7304694, 2296.4172094808905, 2301.1670951932197, 2301.6070472606743, 2302.440
878719799, 2302.6126465387097, 2304.7689688990522, 2306.622205737212, 2307.86
06977025283, 2312.3003697616796, 2313.9310707106206, 2313.9315028755714, 231
9.2358655384751, 2322.6086626894339, 2327.1742521779497, 2331.3099322054973,
2332.1963896721904, 2340.3157906573206, 2340.3563831177507, 2340.623207609460
7, 2346.9475920863679, 2347.446272015613, 2351.670257497849, 2352.21852726314
55, 2352.5326777751675, 2354.122129372221, 2356.2497745357982, 2356.350780338
1058, 2359.1212346973607, 2359.3944138274128, 2360.3230711070041, 2362.132934
4471706, 2363.1887355858821, 2363.5904890653119, 2367.5227559624427, 2367.751
6761687657, 2369.809697001006, 2373.4611856948495, 2376.7351556284093, 2378.2
832043303843, 2379.0311053031651, 2379.6363167509444, 2380.5159104698291, 238
0.68708569606, 2382.1542771197669, 2386.6736266192743, 2387.8796033301178, 23
88.1272997895235, 2388.756998943174, 2389.7449654722573, 2390.2121244776581,
2397.2502998226946, 2398.5112048935689, 2412.5977700395897, 2415.177840242825
8, 2419.8987582128307, 2420.06094964569, 2422.4646540249046, 2423.27010463134
32, 2423.9766913070762, 2428.8149373717215, 2430.8428579404303, 2431.15548659
48001, 2433.1181640027266, 2434.2191766560381, 2438.0621403073383, 2440.58435
62556897, 2442.9555869888427, 2442.9559963290376, 2443.1559508144378, 2443.84
38984517811, 2450.9920440507349, 2451.1560537836021, 2454.7816603518936, 245
5.2908178055, 2455.3343967777587, 2456.6955448325298, 2458.7931999255243, 245
8.9869865454757, 2461.2823893247196, 2461.5235119738345, 2464.2544511474462,
2471.0633338706639, 2474.6840202336944, 2478.3222147251154, 2483.860100730312
8, 2483.8776137322066, 2490.8062951582565, 2492.0963063252593, 2495.434030384
2937, 2499.9095983655088, 2502.1870433682611, 2502.5852632827518, 2502.609637
9579458, 2502.8657574868053, 2506.4530715734536, 2509.0133518975144, 2511.481
0371571593, 2518.9033328017968, 2520.7127960162379, 2522.5156094660742, 2525.
8758876872789, 2527.0003957261265, 2527.5327891048219, 2530.2148130148948, 25
32.6174207724307, 2533.0635601974145, 2543.4044114139615, 2546.1133124823805,
2552.3508771326879, 2552.7195302265386, 2553.7842508716353, 2558.68794502182
3, 2560.2876791485755, 2563.5032670156675, 2566.0278642290696, 2567.478919095
5394, 2571.1048208892612, 2574.6512385175588, 2577.1457079490092, 2579.709479
7670529, 2585.8859216910555, 2586.3420114130304, 2589.7810718282731, 2591.747
6729033633, 2594.6564319770741, 2594.871480439831, 2601.6464018002139, 2602.9
925086330923, 2611.0503633595426, 2612.5263635033425, 2613.9764727326833, 261
4.4410492493421, 2616.7720573255897, 2617.7173262214542, 2617.8888440879227,
2619.0198548311923, 2619.4356262370716, 2620.8624534683236, 2624.006097553890
9, 2624.2821875705363, 2625.3472151317433, 2626.7843459256414, 2628.644517617
3975, 2629.106121859671, 2629.2272629044451, 2631.4098882538237, 2633.3226160
```

```
119461, 2636.6150648132161, 2636.9941221019058, 2648.0156721590602, 2654.6417
837440895, 2660.9274698871445, 2663.6604513338407, 2665.9960615124696, 2672.5
46351328635, 2676.8569255752163, 2678.3175689226996, 2680.0701483356738, 268
0.4908132653618, 2684.5671904424371, 2696.4560074290107, 2700.2168431442688,
2701.5752812016917, 2713.9817611767403, 2721.7273926681196, 2723.095848478345
5, 2731.2899882656179, 2732.5833930550043, 2741.2354878776832, 2743.941872562
1722, 2745.8530550632167, 2746.5860627331522, 2750.1578136536091, 2753.641044
1450062, 2754.4306126675256, 2759.471326178259, 2759.9047084999147, 2771.1761
041117543, 2772.7425051742543, 2773.5695412230066, 2775.7598959564207, 2775.7
793500204589, 2781.0983441798676, 2791.7922558815153, 2794.0039370051004, 279
5.0976727119933, 2796.0545416711743, 2800.7981005420579, 2807.382410716431, 2
809.3216263005556, 2812.1390435040726, 2823.6076214658437, 2826.689406355074,
2867.8120579982224, 2868.4288382318291, 2871.9287943819222, 2887.04502909116
4, 2896.7628484223555, 2902.5773030188188, 2911.4766013141852, 2952.536197915
2772, 2964.5854684930237, 2998.7819193799337, 3018.1804121026298]
```

In [59]:
```python
# scratch cell
#returns the index in distances list that is the lowest value (smallest distan
ce between first image and other images)
#the image which is the most similar to the first image (we can't obviously in
clude that image itself)
def most_similar_index():
    most_similar_index = distances.index(sorted(distances[:500])[1])
    return most_similar_index
index = most_similar_index()
index
```

Out[59]: 61

Then copy the solution you found (an index value) and replace the -1 in the function `most_similar_image` with this value.

In [60]:
```python
# GRADED FUNCTION: DO NOT EDIT THIS LINE
#basically up till now what we have is the images list with the first 500 imag
es
#and the distances list (of size 250,000), which has the distances between all
500 images and itself

#we're computing on the first image compared with the rest of the images
#and at the moment we have the image that is the most similar to the first ima
ge of the dataset (ofc excluding itself)
def most_similar_image():
    """Find the index of the digit, among all MNIST digits
       that is the second-closest to the first image in the dataset (the first
image is closest to itself trivially).
       Your answer should be a single integer.
    """
    index = 61 #<-- Change the -1 to the index of the most similar image.
    # You should do your computation outside this function and update this num
ber
    # once you have computed the result
    return index
```

In [61]:
```python
result = most_similar_image()
```

For the second question, we can compute a `mean` image for each class of image, i.e. we compute mean image for digits of `1`, `2`, `3`,..., `9`, then we compute pairwise distance between them. We can organize the pairwise distances in a 2D plot, which would allow us to visualize the dissimilarity between images of different classes.

First we compute the mean for digits of each class.

In [70]:
```
means = {}
#means of images of each class (1-9)
for n in np.unique(labels):
    means[n] = np.mean(images[labels==n], axis=0)
```

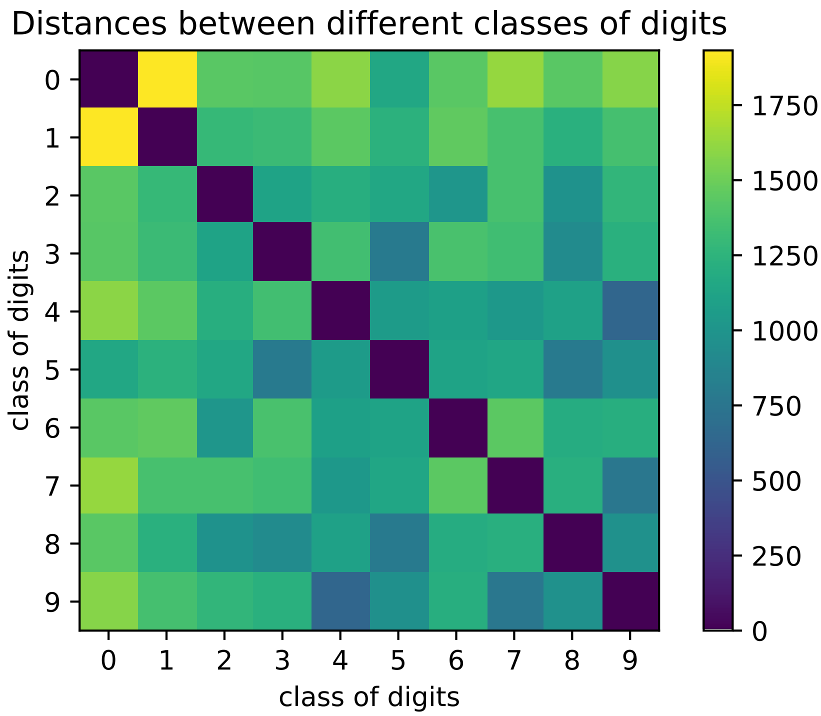
For each pair of classes, we compute the pairwise distance and store them into MD (mean distances). We store the angles between the mean digits in AG

In [63]:
```
MD = np.zeros((10, 10))
AG = np.zeros((10, 10))
#loop through the means (mean of digits of each class(1,2,3 etc...))
#and obtain the distance and angle between each mean
for i in means.keys():
    for j in means.keys():
        MD[i, j] = distance(means[i], means[j])
        AG[i, j] = angle(means[i].ravel(), means[j].ravel())
```

```
/opt/conda/lib/python3.6/site-packages/ipykernel_launcher.py:12: RuntimeWarni
ng: invalid value encountered in arccos
  if sys.path[0] == '':
```

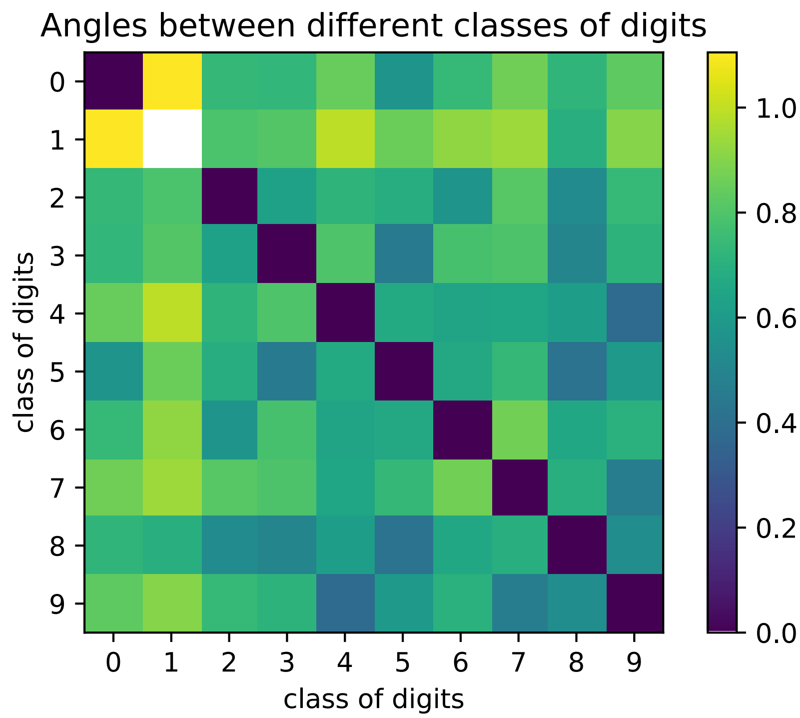
Now we can visualize the distances! Here we put the pairwise distances. The colorbar shows how the distances map to color intensity.

In [75]:
```
fig, ax = plt.subplots()
grid = ax.imshow(MD, interpolation='nearest')
ax.set(title='Distances between different classes of digits',
       xticks=range(10),
       xlabel='class of digits',
       ylabel='class of digits',
       yticks=range(10))
fig.colorbar(grid)
plt.show()
```



Similarly for the angles.

In [18]:
```
fig, ax = plt.subplots()
grid = ax.imshow(AG, interpolation='nearest')
ax.set(title='Angles between different classes of digits',
       xticks=range(10),
       xlabel='class of digits',
       ylabel='class of digits',
       yticks=range(10))
fig.colorbar(grid)
plt.show();
```



K Nearest Neighbors

In this section, we will explore the KNN classification algorithm (https://en.wikipedia.org/wiki/K-nearest_neighbors_algorithm). A classification algorithm takes input some data and use the data to determine which class (category) this piece of data belongs to.

As a motivating example, consider the iris flower dataset (https://archive.ics.uci.edu/ml/datasets/iris). The dataset consists of 150 data points where each data point is a feature vector $\boldsymbol{x} \in \mathbb{R}^4$ describing the attribute of a flower in the dataset, the four dimensions represent

1. sepal length in cm
2. sepal width in cm
3. petal length in cm
4. petal width in cm

and the corresponding target $y \in \mathbb{Z}$ describes the class of the flower. It uses the integers 0, 1 and 2 to represent the 3 classes of flowers in this dataset.

1. Iris Setosa
2. Iris Versicolour
3. Iris Virginica

In [78]:
```
from matplotlib.colors import ListedColormap
from sklearn import neighbors, datasets
iris = datasets.load_iris()
print('data shape is {}'.format(iris.data.shape))
print('class shape is {}'.format(iris.target.shape))
print(iris.data)  #data of all iris's
print(iris.target)   #the outputs/ labels of the iris (either 0 1 or 2)
```

```
data shape is (150, 4)
class shape is (150,)
[[ 5.1  3.5  1.4  0.2]
 [ 4.9  3.   1.4  0.2]
 [ 4.7  3.2  1.3  0.2]
 [ 4.6  3.1  1.5  0.2]
 [ 5.   3.6  1.4  0.2]
 [ 5.4  3.9  1.7  0.4]
 [ 4.6  3.4  1.4  0.3]
 [ 5.   3.4  1.5  0.2]
 [ 4.4  2.9  1.4  0.2]
 [ 4.9  3.1  1.5  0.1]
 [ 5.4  3.7  1.5  0.2]
 [ 4.8  3.4  1.6  0.2]
 [ 4.8  3.   1.4  0.1]
 [ 4.3  3.   1.1  0.1]
 [ 5.8  4.   1.2  0.2]
 [ 5.7  4.4  1.5  0.4]
 [ 5.4  3.9  1.3  0.4]
 [ 5.1  3.5  1.4  0.3]
 [ 5.7  3.8  1.7  0.3]
 [ 5.1  3.8  1.5  0.3]
 [ 5.4  3.4  1.7  0.2]
 [ 5.1  3.7  1.5  0.4]
 [ 4.6  3.6  1.   0.2]
 [ 5.1  3.3  1.7  0.5]
 [ 4.8  3.4  1.9  0.2]
 [ 5.   3.   1.6  0.2]
 [ 5.   3.4  1.6  0.4]
 [ 5.2  3.5  1.5  0.2]
 [ 5.2  3.4  1.4  0.2]
 [ 4.7  3.2  1.6  0.2]
 [ 4.8  3.1  1.6  0.2]
 [ 5.4  3.4  1.5  0.4]
 [ 5.2  4.1  1.5  0.1]
 [ 5.5  4.2  1.4  0.2]
 [ 4.9  3.1  1.5  0.1]
 [ 5.   3.2  1.2  0.2]
 [ 5.5  3.5  1.3  0.2]
 [ 4.9  3.1  1.5  0.1]
 [ 4.4  3.   1.3  0.2]
 [ 5.1  3.4  1.5  0.2]
 [ 5.   3.5  1.3  0.3]
 [ 4.5  2.3  1.3  0.3]
 [ 4.4  3.2  1.3  0.2]
 [ 5.   3.5  1.6  0.6]
 [ 5.1  3.8  1.9  0.4]
 [ 4.8  3.   1.4  0.3]
 [ 5.1  3.8  1.6  0.2]
 [ 4.6  3.2  1.4  0.2]
 [ 5.3  3.7  1.5  0.2]
 [ 5.   3.3  1.4  0.2]
 [ 7.   3.2  4.7  1.4]
 [ 6.4  3.2  4.5  1.5]
 [ 6.9  3.1  4.9  1.5]
 [ 5.5  2.3  4.   1.3]
 [ 6.5  2.8  4.6  1.5]
```

```
[ 5.7  2.8  4.5  1.3]
[ 6.3  3.3  4.7  1.6]
[ 4.9  2.4  3.3  1. ]
[ 6.6  2.9  4.6  1.3]
[ 5.2  2.7  3.9  1.4]
[ 5.   2.   3.5  1. ]
[ 5.9  3.   4.2  1.5]
[ 6.   2.2  4.   1. ]
[ 6.1  2.9  4.7  1.4]
[ 5.6  2.9  3.6  1.3]
[ 6.7  3.1  4.4  1.4]
[ 5.6  3.   4.5  1.5]
[ 5.8  2.7  4.1  1. ]
[ 6.2  2.2  4.5  1.5]
[ 5.6  2.5  3.9  1.1]
[ 5.9  3.2  4.8  1.8]
[ 6.1  2.8  4.   1.3]
[ 6.3  2.5  4.9  1.5]
[ 6.1  2.8  4.7  1.2]
[ 6.4  2.9  4.3  1.3]
[ 6.6  3.   4.4  1.4]
[ 6.8  2.8  4.8  1.4]
[ 6.7  3.   5.   1.7]
[ 6.   2.9  4.5  1.5]
[ 5.7  2.6  3.5  1. ]
[ 5.5  2.4  3.8  1.1]
[ 5.5  2.4  3.7  1. ]
[ 5.8  2.7  3.9  1.2]
[ 6.   2.7  5.1  1.6]
[ 5.4  3.   4.5  1.5]
[ 6.   3.4  4.5  1.6]
[ 6.7  3.1  4.7  1.5]
[ 6.3  2.3  4.4  1.3]
[ 5.6  3.   4.1  1.3]
[ 5.5  2.5  4.   1.3]
[ 5.5  2.6  4.4  1.2]
[ 6.1  3.   4.6  1.4]
[ 5.8  2.6  4.   1.2]
[ 5.   2.3  3.3  1. ]
[ 5.6  2.7  4.2  1.3]
[ 5.7  3.   4.2  1.2]
[ 5.7  2.9  4.2  1.3]
[ 6.2  2.9  4.3  1.3]
[ 5.1  2.5  3.   1.1]
[ 5.7  2.8  4.1  1.3]
[ 6.3  3.3  6.   2.5]
[ 5.8  2.7  5.1  1.9]
[ 7.1  3.   5.9  2.1]
[ 6.3  2.9  5.6  1.8]
[ 6.5  3.   5.8  2.2]
[ 7.6  3.   6.6  2.1]
[ 4.9  2.5  4.5  1.7]
[ 7.3  2.9  6.3  1.8]
[ 6.7  2.5  5.8  1.8]
[ 7.2  3.6  6.1  2.5]
[ 6.5  3.2  5.1  2. ]
[ 6.4  2.7  5.3  1.9]
```

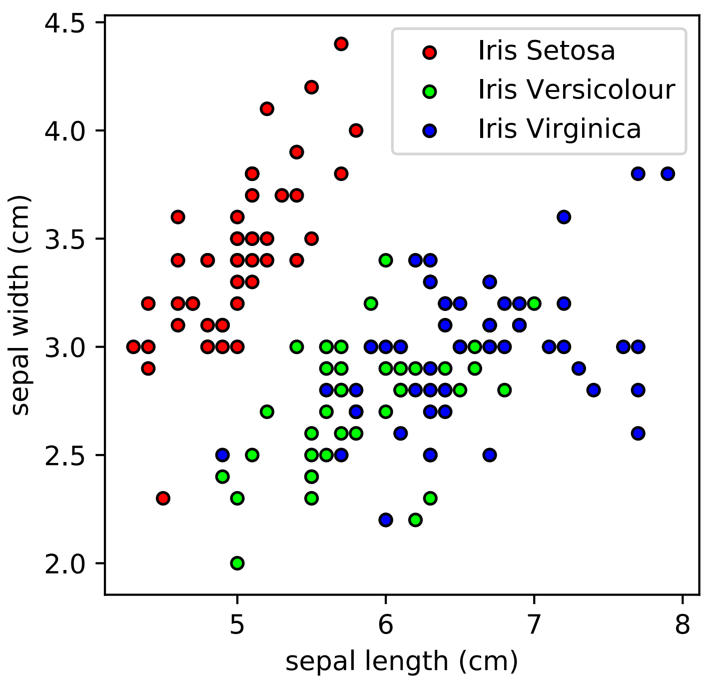
```
 [ 6.8  3.   5.5  2.1]
 [ 5.7  2.5  5.   2. ]
 [ 5.8  2.8  5.1  2.4]
 [ 6.4  3.2  5.3  2.3]
 [ 6.5  3.   5.5  1.8]
 [ 7.7  3.8  6.7  2.2]
 [ 7.7  2.6  6.9  2.3]
 [ 6.   2.2  5.   1.5]
 [ 6.9  3.2  5.7  2.3]
 [ 5.6  2.8  4.9  2. ]
 [ 7.7  2.8  6.7  2. ]
 [ 6.3  2.7  4.9  1.8]
 [ 6.7  3.3  5.7  2.1]
 [ 7.2  3.2  6.   1.8]
 [ 6.2  2.8  4.8  1.8]
 [ 6.1  3.   4.9  1.8]
 [ 6.4  2.8  5.6  2.1]
 [ 7.2  3.   5.8  1.6]
 [ 7.4  2.8  6.1  1.9]
 [ 7.9  3.8  6.4  2. ]
 [ 6.4  2.8  5.6  2.2]
 [ 6.3  2.8  5.1  1.5]
 [ 6.1  2.6  5.6  1.4]
 [ 7.7  3.   6.1  2.3]
 [ 6.3  3.4  5.6  2.4]
 [ 6.4  3.1  5.5  1.8]
 [ 6.   3.   4.8  1.8]
 [ 6.9  3.1  5.4  2.1]
 [ 6.7  3.1  5.6  2.4]
 [ 6.9  3.1  5.1  2.3]
 [ 5.8  2.7  5.1  1.9]
 [ 6.8  3.2  5.9  2.3]
 [ 6.7  3.3  5.7  2.5]
 [ 6.7  3.   5.2  2.3]
 [ 6.3  2.5  5.   1.9]
 [ 6.5  3.   5.2  2. ]
 [ 6.2  3.4  5.4  2.3]
 [ 5.9  3.   5.1  1.8]]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2]
```

For the simplicity of the exercise, we will only use the first 2 dimensions (sepal length and sepal width) of as features used to classify the flowers.

In [81]:
```
X = iris.data[:, :2] # use first two version for simplicity
y = iris.target
```

We create a scatter plot of the dataset below. The x and y axis represent the sepal length and sepal width of the dataset, and the color of the points represent the different classes of flowers.

In [82]:
```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap
from sklearn import neighbors, datasets
iris = datasets.load_iris()
cmap_light = ListedColormap(['#FFAAAA', '#AAFFAA', '#AAAAFF'])
cmap_bold = ListedColormap(['#FF0000',  '#00FF00', '#0000FF'])

K = 3
x = X[-1]

fig, ax = plt.subplots(figsize=(4,4))
for i, iris_class in enumerate(['Iris Setosa', 'Iris Versicolour', 'Iris Virgi
nica']):
    idx = y==i
    ax.scatter(X[idx,0], X[idx,1],
               c=cmap_bold.colors[i], edgecolor='k',
               s=20, label=iris_class);
ax.set(xlabel='sepal length (cm)', ylabel='sepal width (cm)')
ax.legend();
```



The idea behind a KNN classifier is pretty simple: Given a training set $\boldsymbol{X} \in \mathbb{R}^{N \times D}$ and $\boldsymbol{y} \in \mathbb{Z}^N$, we predict the label of a new point $\boldsymbol{x} \in \mathbb{R}^D$ **as the label of the majority of its "K nearest neighbor"** (hence the name KNN) by some distance measure (e.g the Euclidean distance). Here, N is the number of data points in the dataset, and D is the dimensionality of the data.

In [117]:
```
# GRADED FUNCTION: DO NOT EDIT THIS LINE

def pairwise_distance_matrix(X, Y):
    """Compute the pairwise distance between rows of X and rows of Y

    Arguments
    ----------
    X: ndarray of size (N, D)
    Y: ndarray of size (M, D)

    Returns
    --------
    distance_matrix: matrix of shape (N, M), each entry distance_matrix[i,j] i
s the distance between
    ith row of X and the jth row of Y (we use the dot product to compute the d
istance).
    """
    N, D = X.shape
    M, _ = Y.reshape(-1,D).shape
#     print(N, M, D) #2 2 2
    print(X.shape)
    print(X[:,np.newaxis,:].shape)

    print(Y.shape)
    print(Y[np.newaxis,...].shape)

    #np.newaxix adds a new axis, X becomes (N, 1, D), Y (1, M, D), upon subtra
ction matrix (N, M, D) results
    #The entry of result matrix is the vector of i row of X minus j row of Y
    distance_matrix = np.sqrt(np.sum((X[:,np.newaxis,:] - Y[np.newaxis,...])**
2, axis=2))
    return distance_matrix

pairwise_distance_matrix(np.array([[1, 2], [3, 4]]), np.array([[5, 6], [7, 8
]]))
```

```
(2, 2)
(2, 1, 2)
(2, 2)
(1, 2, 2)
```

Out[117]:
```
array([[ 5.65685425,  8.48528137],
       [ 2.82842712,  5.65685425]])
```

For `pairwise_distance_matrix`, you may be tempting to iterate through rows of $\boldsymbol{X}$ and $\boldsymbol{Y}$ and fill in the distance matrix, but that is slow! Can you think of some way to vectorize your computation (i.e. make it faster by using numpy/scipy operations only)?

In [109]:
```
# GRADED FUNCTION: DO NOT EDIT THIS LINE

def KNN(k, X, y, x):
    """K nearest neighbors
    k: number of nearest neighbors
    X: training input locations
    y: training labels
    x: test input
    """
    N, D = X.shape
    num_classes = len(np.unique(y))
    dist = pairwise_distance_matrix(X, x)
    # <-- EDIT THIS to compute the pairwise distance matrix

    # Next we make the predictions
    ypred = np.zeros(num_classes)
    classes = y[np.argsort(dist)][:k] # find the labels of the k nearest neigh
bors
    for c in np.unique(classes):
        ypred[c] = len(classes[classes == c])  # <-- EDIT THIS to compute the
 correct prediction

    return np.argmax(ypred)
```

We can also visualize the "decision boundary" of the KNN classifier, which is the region of a problem space in which the output label of a classifier is ambiguous. This would help us develop an intuition of how KNN behaves in practice. The code below plots the decision boundary.

In [110]:
```
x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
step = 0.1
xx, yy = np.meshgrid(np.arange(x_min, x_max, step),
                     np.arange(y_min, y_max, step))

ypred = []
for data in np.array([xx.ravel(), yy.ravel()]).T:
    ypred.append(KNN(K, X, y, data.reshape(1,2)))

fig, ax = plt.subplots(figsize=(4,4))

ax.pcolormesh(xx, yy, np.array(ypred).reshape(xx.shape), cmap=cmap_light)
ax.scatter(X[:,0], X[:,1], c=y, cmap=cmap_bold, edgecolor='k', s=20);
```