ISTA 421/521 – Homework 5

Due: Wednesday, December 5, 5pm

20 pts total for Undergrads, 24 pts total for Grads

Ken Youens-Clark

Graduate

Instructions

In this assignment, exercises 5 and 6 require you to write small python scripts; the details for those scripts, along with their .py name are described in the exercises. All of the exercises in this homework require written derivations, short answers, and/or plots, so you will also submit a .pdf of your written answers. (You can use LaTeX or any other system (including handwritten; plots, of course, must be program-generated) as long as the final version is in PDF.)

The final submission will include (minimally) the two scripts you need to write for problems 5 and 6, and a PDF version of your written part of the assignment. You are required to create either a .zip or tarball (.tar.gz / .tgz) archive of all of the files for your submission and submit your archive to the d2l dropbox by the date/time deadline above.

NOTE: Problem 3 is required for Graduate students only; Undergraduates may complete this problem for extra credit equal to the point value.

(FCMA refers to the course text: Rogers and Girolami (2016), *A First Course in Machine Learning*, second edition. For general notes on using LaTeX to typeset math, see:
`http://en.wikibooks.org/wiki/LaTeX/Mathematics`)

1. [5 points] Adapted from **Exercise 5.3** of FCMA p.202:

   Compute the maximum likelihood estimates of $\boldsymbol{\mu}_c$ and $\boldsymbol{\Sigma}_c$ for class c of a Bayesian classifier with Gaussian class-conditionals and a set of N_c objects belonging to class c, $\mathbf{x}_1$, ..., $\mathbf{x}_{N_c}$.

   Solution.

Likelihood

$$L = p(\mathbf{X}^c|\boldsymbol{\mu}_c, \boldsymbol{\Sigma}_c)$$

is product of (Gaussian) probabilities

$$\begin{aligned}
&= \prod_n^N \mathcal{N}(\mathbf{x}_n|\boldsymbol{\mu}_c, \boldsymbol{\Sigma}_c) \\
&= \prod_n^N \frac{1}{(2\pi)^{N/2}|\boldsymbol{\Sigma}_c|^{1/2}} \exp\left\{-\frac{1}{2}(\mathbf{x}_n \ -\boldsymbol{\mu}_c)^\top \boldsymbol{\Sigma}_c^{-1}(\mathbf{x}_n - \boldsymbol{\mu}_c)\right\} \\
&\propto |\boldsymbol{\Sigma}_c|^{-N_c/2} \exp\left\{\sum_n^{N_c} -\frac{1}{2}(\mathbf{x}_n \ -\boldsymbol{\mu}_c)^\top \boldsymbol{\Sigma}_c^{-1}(\mathbf{x}_n - \boldsymbol{\mu}_c)\right\}
\end{aligned}$$

Use log to handle exponent

$$\log(L) = \left(-\frac{N_c}{2}\log(|\boldsymbol{\Sigma}_c|)\right)\left(-\frac{1}{2}\sum_n^{N_c}(\mathbf{x}_n \ -\boldsymbol{\mu}_c)^\top \boldsymbol{\Sigma}_c^{-1}(\mathbf{x}_n - \boldsymbol{\mu}_c)\right)$$

Take derivative w.r.t. $\boldsymbol{\mu}_c$

$$\begin{aligned}
\frac{\partial \log(L)}{\partial \boldsymbol{\mu}_c} &= -\frac{1}{2}\sum_n^{N_c} 2\boldsymbol{\Sigma}_c^{-1}(\mathbf{x}_n - \boldsymbol{\mu}_c) \\
&= -\sum_n^{N_c} \boldsymbol{\Sigma}_c^{-1}(\mathbf{x}_n - \boldsymbol{\mu}_c) = 0 \\
\sum_n^{N_c} \boldsymbol{\Sigma}_c^{-1}\boldsymbol{\mu}_c - \sum_n^{N_c} \boldsymbol{\Sigma}_c^{-1}\mathbf{x}_n &= 0 \\
\boldsymbol{\Sigma}_c^{-1}N_c\boldsymbol{\mu}_c - \boldsymbol{\Sigma}_c^{-1}\sum_n^{N_c}\mathbf{x}_n &= 0 \\
\boldsymbol{\Sigma}_c^{-1}N_c\boldsymbol{\mu}_c &= \boldsymbol{\Sigma}_c^{-1}\sum_n^{N_c}\mathbf{x}_n \\
N_c\boldsymbol{\mu}_c &= \sum_n^{N_c}\mathbf{x}_n \\
\boldsymbol{\mu}_c &= \frac{1}{N_c}\sum_n^{N}\mathbf{x}_n
\end{aligned}$$

Take derivative w.r.t. $\mathbf{\Sigma}_c$, set to zero, solve

$$\frac{\partial \log(L)}{\partial \mathbf{\Sigma}_c} = -\frac{N_c}{2}(\mathbf{\Sigma}_c^{-1}) + \frac{1}{2}\sum_n^{N_c}(\mathbf{x}_n - \boldsymbol{\mu}_c)^\top \mathbf{\Sigma}_c^{-2}(\mathbf{x}_n - \boldsymbol{\mu}_c) = 0$$

$$\frac{N_c}{2}(\mathbf{\Sigma}_c{}^{-1}) = \frac{1}{2}\sum_n^{N_c}(\mathbf{x}_n - \boldsymbol{\mu}_c)^\top \mathbf{\Sigma}_c^{-2}(\mathbf{x}_n - \boldsymbol{\mu}_c)$$

$$N_c(\mathbf{\Sigma}_c{}^{-1}) = \mathbf{\Sigma}_c^{-1}\left(\sum_n^{N_c}(\mathbf{x}_n - \boldsymbol{\mu}_c)^\top(\mathbf{x}_n - \boldsymbol{\mu}_c)\right)\mathbf{\Sigma}_c^{-1}$$

$$N_c\cancel{(\mathbf{\Sigma}_c{}^{-1})(\mathbf{\Sigma}_c^{-1})} = \mathbf{\Sigma}_c^{-1}\left(\sum_n^{N_c}(\mathbf{x}_n - \boldsymbol{\mu}_c)^\top(\mathbf{x}_n - \boldsymbol{\mu}_c)\right)\cancel{\mathbf{\Sigma}_c^{-1}(\mathbf{\Sigma}_c^{-1})}$$

$$\cancel{\frac{1}{N_c}}N_c(\mathbf{\Sigma}_c{}^{-1}) = \frac{1}{N_c}\cancel{(\mathbf{\Sigma}_c{}^{-1})\mathbf{\Sigma}_c^{-1}}\left(\sum_n^{N_c}(\mathbf{x}_n - \boldsymbol{\mu}_c)^\top(\mathbf{x}_n - \boldsymbol{\mu}_c)\right)$$

$$\mathbf{\Sigma}_c{}^{-1} = \frac{1}{N_c}\left(\sum_n^{N_c}(\mathbf{x}_n - \boldsymbol{\mu}_c)^\top(\mathbf{x}_n - \boldsymbol{\mu}_c)\right)$$

2. [4 points] Adapted from **Exercise 5.4** of FCMA p.204:

   Compute the maximum likelihood estimates of q_{mc} for class c of a Bayesian classifier with multinomial class-conditionals and a set of N_c, M-dimensional objects belonging to class c: $\mathbf{x}_1, ..., \mathbf{x}_{N_c}$.

   Solution.

$$L = P(x_n|\mathbf{q})$$
$$= \left(\frac{s_n!}{\prod_{m=1}^M x_{nm}!}\right)\prod_{m=1}^M q_m{}^{x_{nm}}$$

   The answer

$$q_{cm} = \frac{\sum_{n=1}^{N_c} x_{nm}}{\sum_{m'=1}^{M}\sum_{n=1}^{N_c} x_{nm'}}$$

3. [4 points] **Required only for Graduates** Adapted from **Exercise 5.5** of FCMA p.204:

   For a Bayesian classifier with multinomial class-conditionals with M-dimensional parameters $\mathbf{q}_c$, compute the posterior Dirichlet for class c when the prior over $\mathbf{q}_c$ is a Dirichlet with constant parameter α and the observations belonging to class c are the N_c observations $\mathbf{x}_1, ..., \mathbf{x}_{N_c}$.

   Solution.

$$L(\mathbf{X}, \mathbf{q}) \propto \prod_{m=1}^M q_m^{x_{nm}}$$

4. [3 points] For a support vector machine, if we remove one of the support vectors from the training set, does the size of the maximum margin decrease, stay the same, or increase for that dataset? Why? Also

justify your answer by providing a simple dataset (no more than 2-dimensions) in which you identify the support vectors, draw the location of the maximum margin hyperplane, remove one of the support vectors, and draw the location of the resulting maximum margin hyperplane. Drawing this by hand is sufficient.

Solution.

5. [4 points] In this exercise you will use the python script, `knn.py`, which contains code to plot the decision boundary for a k-nearest neighbors classifier. Two data sources are also provided in the `data` directory: `knn_binary_data.csv` and `knn_three_class_data.csv`.

   In python file, the function `knn` is to compute the k-nearest neighbors classification for a point `p`, but the functionality is not currently implemented. You *can* run the code in its unimplemented state, but the decision boundary will not display (everything will be considered the same class). You must implement this yourself. Use a Euclidean distance measure for determining the neighbors (note: you don' t need to take the square root! The sum of squares is sufficient). You do not need to implement any fancy indexing – you can simply search for the k nearest points by searching over the pairwise distance of the input (`p`) to all of the "training" inputs (`x`). Use the maximum class frequency as the class decision rule. You do not need to worry about doing anything special for breaking ties. The numpy function `argsort` and the python `collections.Counter` may be of use, but are not required. Submit the `knn.py` file with your implementation. In your written solution, run the code on both of the provided data sources (`knn_binary_data.csv` and `knn_three_class_data.csv`.), and for each, plot the decision boundary for $k = \{1, 5, 10, 59\}$. Include informative captions and describe any patterns you see in the decision boundaries as k changes.

   Solution.

   When k=1, the boundaries are too close, creating islands of the green squares when they ought to be assumed to have crossed into enemy territory. K of 5 or 10 does a much better job capturing what I would judge to be the real boundary. K of 59 is far too high and pushes the border too far into green square's territory because there are fewer of those class than the other.

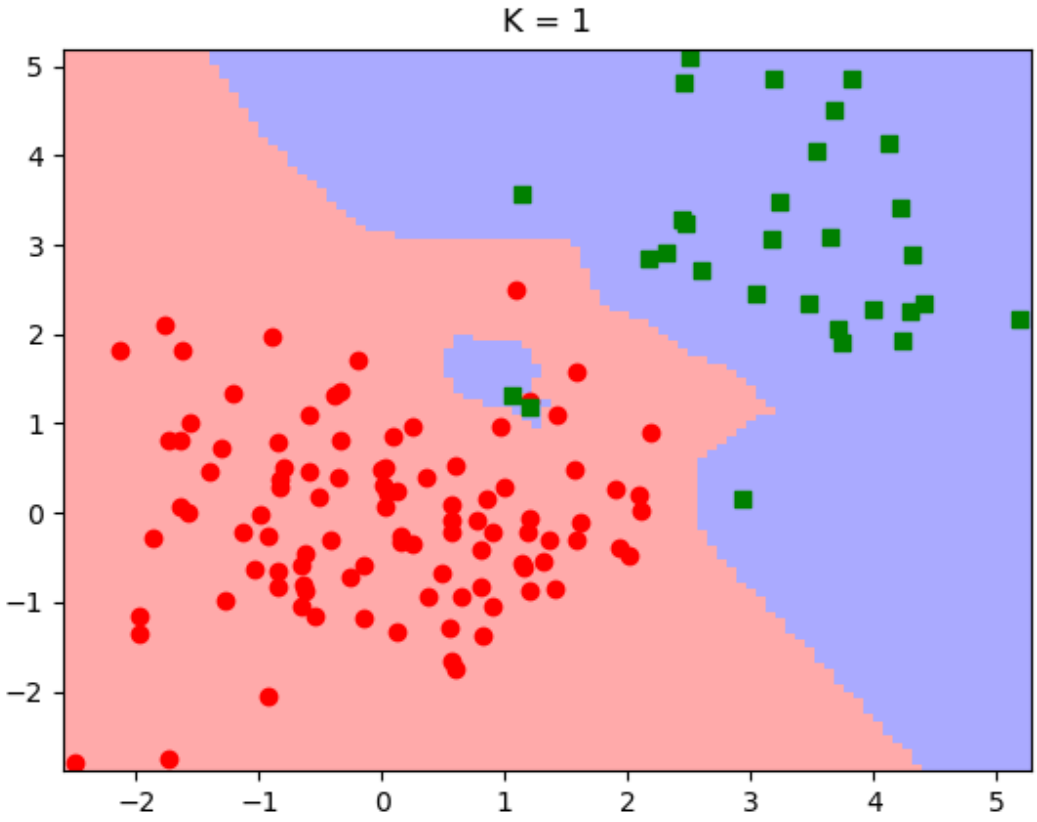

Figure 1: KNN k=1 binary classification

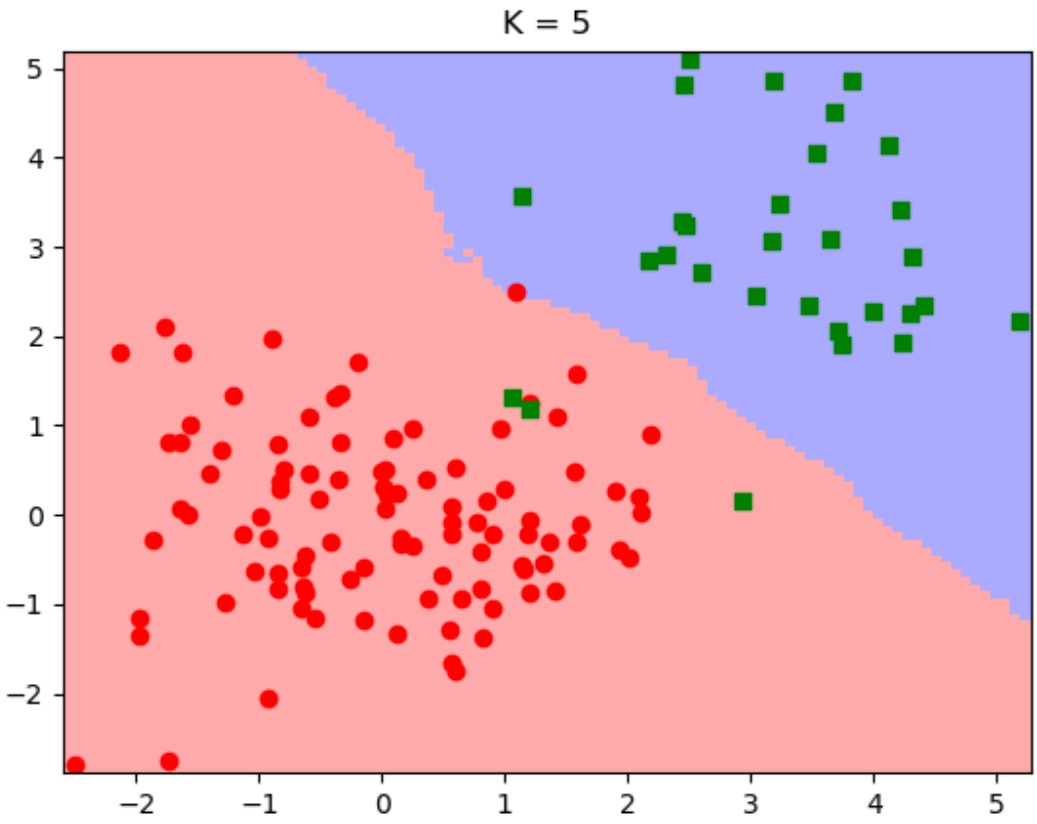

Figure 2: KNN k=5 binary classification

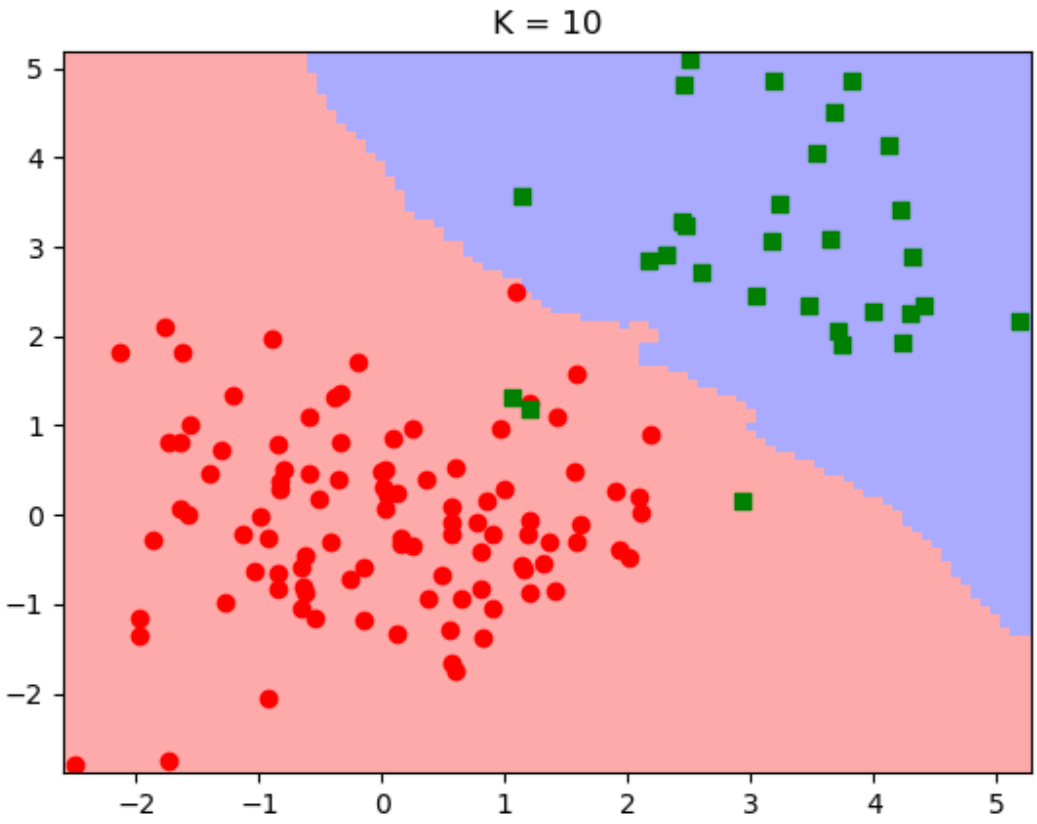

Figure 3: KNN k=10 binary classification

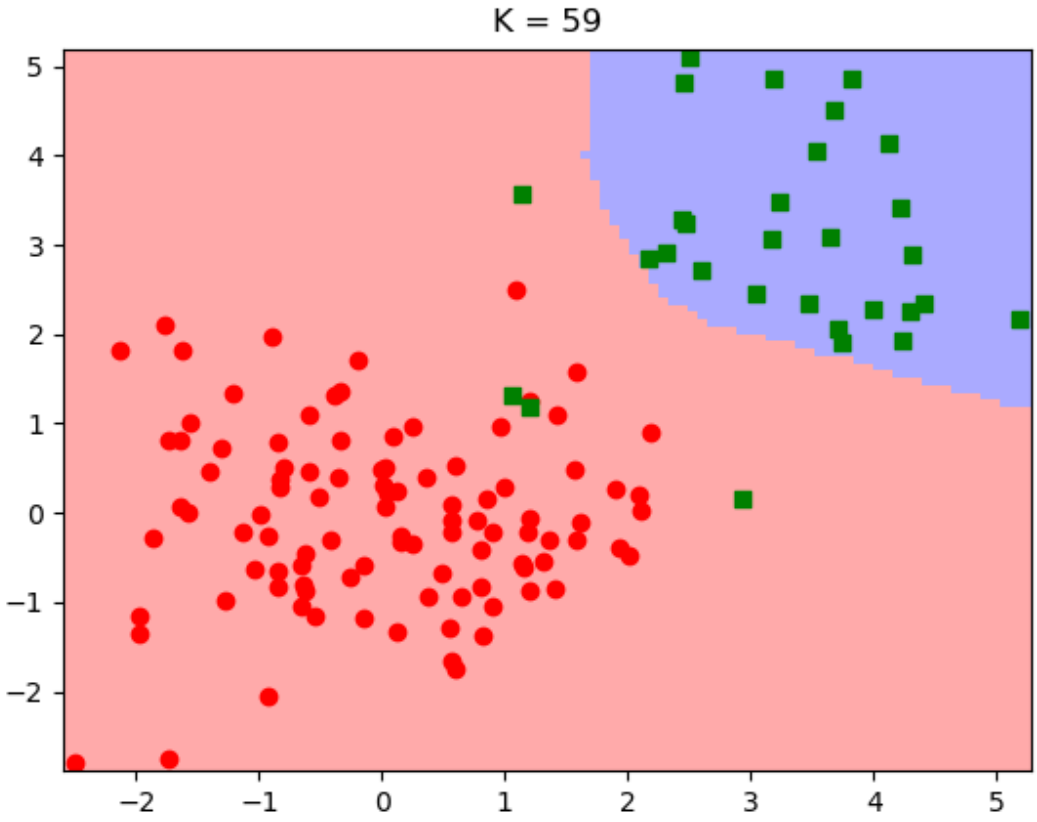

Figure 4: KNN k=59 binary classification

For the classification of three classes, k of 1 again is too liberal with creating boundaries for instance the one green square that is clearly in the red circle's area and also a red circle that ventures into the blue region. Again k of 5 and 10 create smoother boundaries that allow errant positions to wander where they ought not. K of 59 isn't actually terrible since there are roughly equal numbers of the three classes and so no one class gets marginalized too much, but it does a poorer job than 5 or 10.

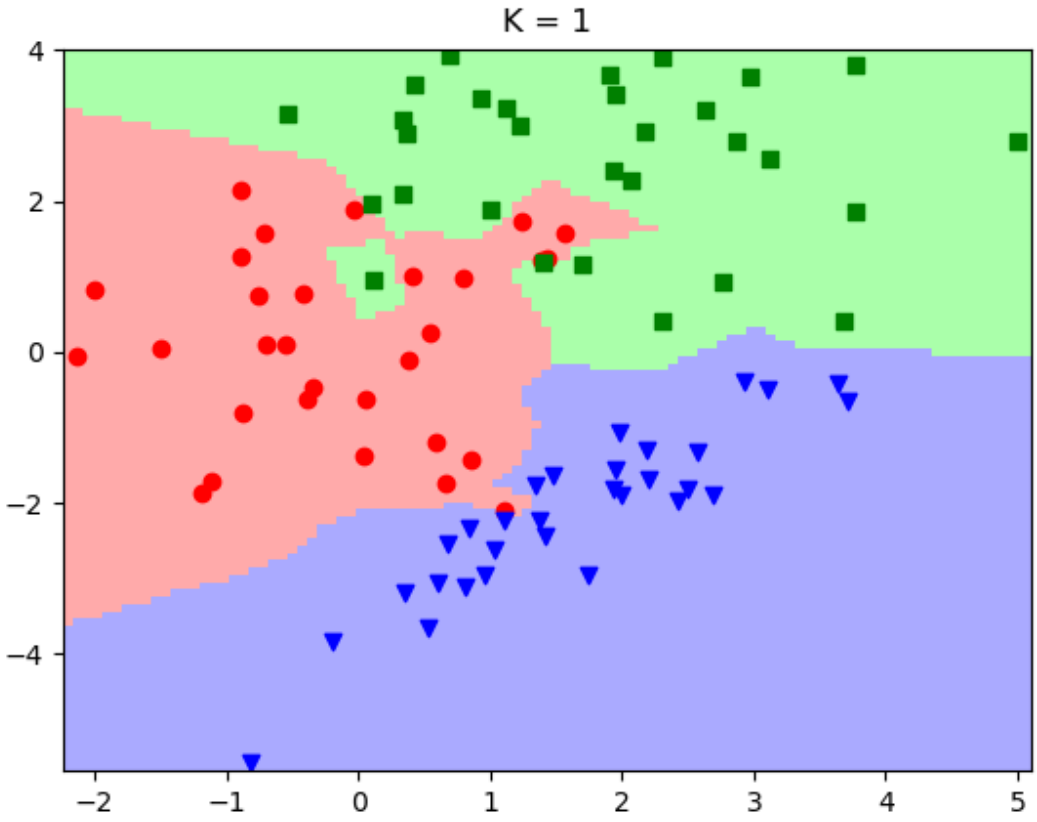

Figure 5: KNN k=1 three class classification

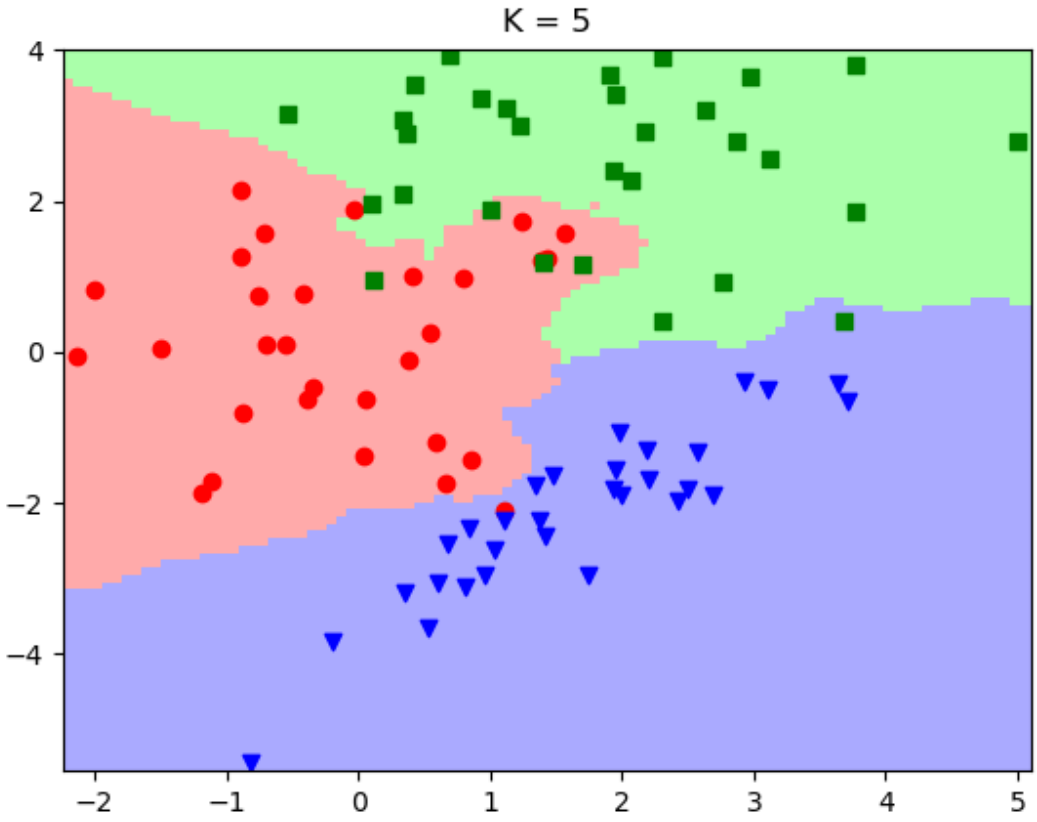

Figure 6: KNN k=5 three class classification

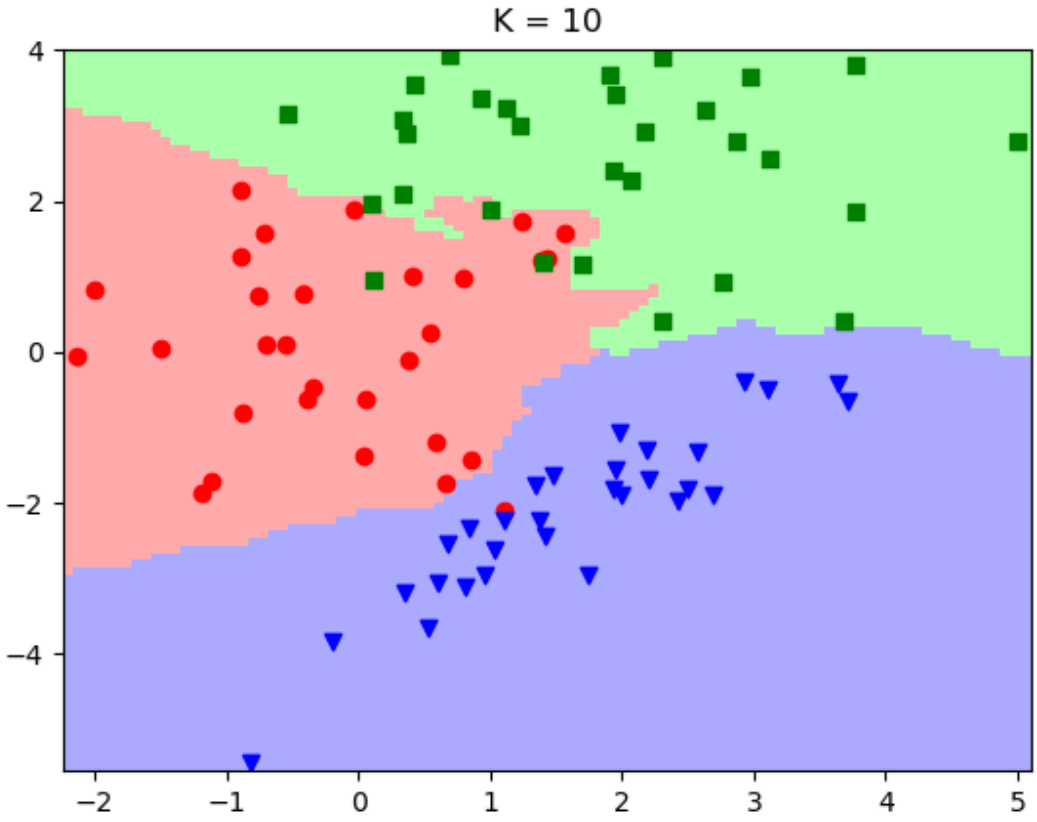

Figure 7: KNN k=10 three class classification

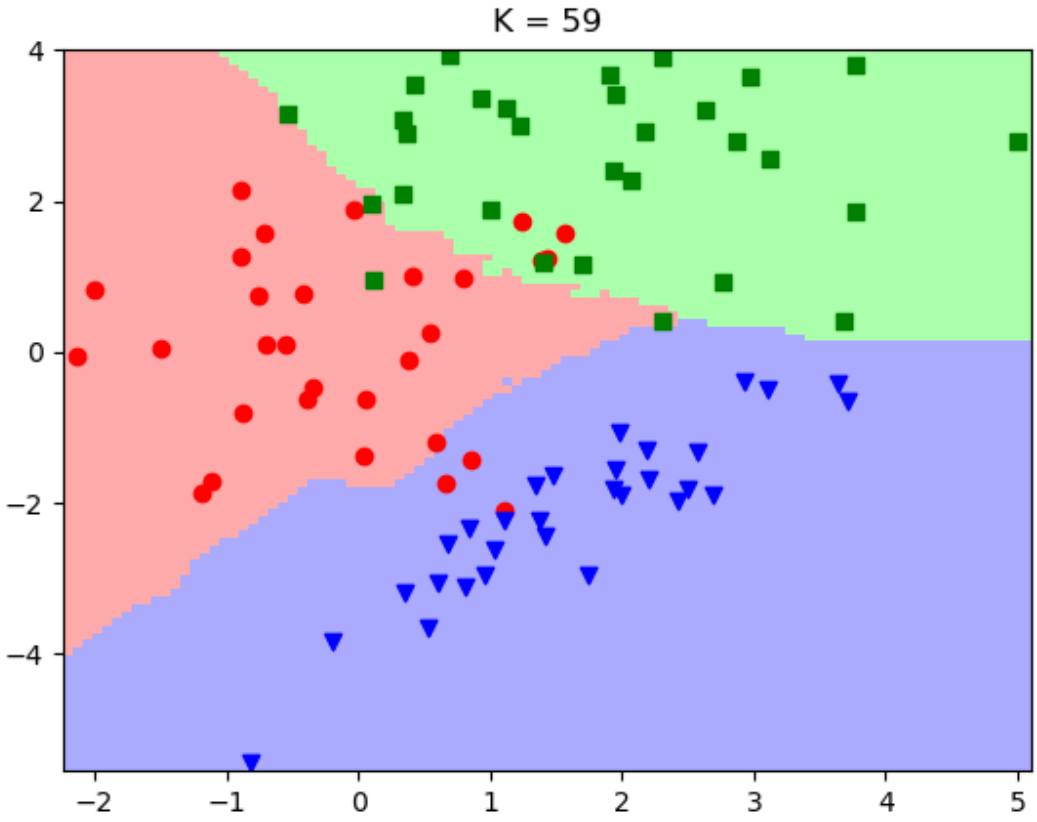

Figure 8: KNN k=59 three class classification

```
#!/usr/bin/env python3
"""
Author : kyclark
Date   : 2018-11-24
Purpose: K-Nearest Neighbors
"""

import argparse
import matplotlib
import os
#matplotlib.use('TkAgg')
import matplotlib.pyplot as plt
import numpy as np
import sys
from collections import Counter
from matplotlib.colors import ListedColormap
from scipy.spatial import distance
```

```
# --------------------------------------------------
def get_args():
    """get command-line arguments"""
    parser = argparse.ArgumentParser(
        description='K-Nearest Neighbors',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument(
        '-f',
        '--file',
        metavar='FILE',
        help='Input file',
        default='../data/knn_binary_data.csv')

    parser.add_argument(
        '-k',
        metavar='INT',
        help='Values for K',
        nargs='+',
        type=int,
        default=[1, 5, 10, 59])

    parser.add_argument(
        '-o',
        '--out_dir',
        help='Output directory for saved figures',
        metavar='DIR',
        type=str,
        default=None)

    parser.add_argument(
        '-g',
        '--granularity',
        help='Granularity',
        metavar='int',
        type=int,
        default=100)

    parser.add_argument(
        '-q', '--quiet', help='Do not show figures', action='store_true')

    return parser.parse_args()


# --------------------------------------------------
def warn(msg):
    """Print a message to STDERR"""
    print(msg, file=sys.stderr)


# --------------------------------------------------
```

```
def die(msg='Something bad happened'):
    """warn() and exit with error"""
    warn(msg)
    sys.exit(1)


# ---------------------------------------------------
def read_data(path, d=','):
    """
    Read 2-dimensional real-valued features with associated class labels
    :param path: path to csv file
    :param d: delimiter
    :return: x=array of features, t=class labels
    """
    arr = np.genfromtxt(path, delimiter=d, dtype=None)
    length = len(arr)
    x = np.zeros(shape=(length, 2))
    t = np.zeros(length)
    for i, (x1, x2, tv) in enumerate(arr):
        x[i, 0] = x1
        x[i, 1] = x2
        t[i] = int(tv)
    return x, t


# ---------------------------------------------------
def knn(p, k, x, t):
    """
    K-Nearest Neighbors classifier.  Return the most frequent class among the k
    nearest points
    :param p: point to classify (assumes 2-dimensional)
    :param k: number of nearest neighbors
    :param x: array of observed 2-dimensional points
    :param t: array of target labels (corresponding to points)
    :return: the top class label
    """

    d = np.argsort(list(map(lambda z: distance.euclidean(p, z), x)))[:k]
    count = Counter(t[d])

    # most_common() returns a sorted list of tuples
    # so take the first element of the first tuple -- [0][0]
    return count.most_common(1)[0][0]


# ---------------------------------------------------
def plot_decision_boundary(k, x, t, granularity=100, out_file=None):
    """
    Given data (observed x and labels t) and choice k of nearest neighbors,
    plots the decision boundary based on a grid of classifications over the
    feature space.
```

```
:param k: number of nearest neighbors
:param x: array of observed 2-dimensional points
:param t: array of target labels (corresponding to points)
:param granularity: controls granularity of the meshgrid
:return:
"""
print('KNN for K={0}'.format(k))

# Initialize meshgrid to be used to store the class prediction values
# this is used for computing and plotting the decision boundary contour
Xv, Yv = np.meshgrid(
    np.linspace(np.min(x[:, 0]) - 0.1,
                np.max(x[:, 0]) + 0.1, granularity),
    np.linspace(np.min(x[:, 1]) - 0.1,
                np.max(x[:, 1]) + 0.1, granularity))

# Calculate KNN classification for every point in meshgrid
classes = np.zeros(shape=(Xv.shape[0], Xv.shape[1]))
for i in range(Xv.shape[0]):
    for j in range(Xv.shape[1]):
        classes[i][j] = knn(np.array([Xv[i][j], Yv[i][j]]), k, x, t)

# plot the binary decision boundary contour
plt.figure()

# Create color map
cmap_light = ListedColormap(['#FFAAAA', '#AAFFAA', '#AAAAFF'])
plt.pcolormesh(Xv, Yv, classes, cmap=cmap_light)
ti = 'K = {0}'.format(k)
plt.title(ti)
plt.draw()

# Plot the points
ma = ['o', 's', 'v']
fc = ['r', 'g', 'b']  # np.array([0, 0, 0]), np.array([1, 1, 1])]
tv = np.unique(t.flatten())  # an array of the unique class labels

#if new_figure:
#    plt.figure()

for i in range(tv.shape[0]):
    # returns a boolean vector mask for selecting just the instances of class tv[i]
    pos = (t == tv[i]).nonzero()
    plt.scatter(
        np.asarray(x[pos, 0]),
        np.asarray(x[pos, 1]),
        marker=ma[i],
        facecolor=fc[i])

if out_file:
    warn('Saving figure to "{}"'.format(out_file))
```

```
        plt.savefig(out_file)


# ------------------------------------------------
def main():
    args = get_args()
    in_file = args.file
    K = args.k
    granularity = args.granularity
    out_dir = args.out_dir

    if not os.path.isfile(in_file):
        die('"{}" is not a file'.format(in_file))

    x, t = read_data(in_file)

    basename, _ = os.path.splitext(os.path.basename(in_file))

    if out_dir:
        out_dir = os.path.abspath(out_dir)

    # Loop over different neighborhood values K
    for k in K:
        out_file = None
        if out_dir:
            out_file = os.path.join(out_dir, '{}-k-{}.png'.format(basename, k))
        plot_decision_boundary(
            k, x, t, granularity=granularity, out_file=out_file)

    if not args.quiet:
        warn('Showing figures')
        plt.show()

    warn('Done')


# ------------------------------------------------
if __name__ == '__main__':
    main()
```

6. [4 points] Using your implementation of your KNN classifier in exercise 5, write a script to perform 10-fold cross-validation in a search for the best choice of K. Remember to randomize your data at the start of the CV procedure, but use the same CV folds for each K. Make your script search in the range of $1 \leq K \leq 30$. Run your script on both data sources: `knn_binary_data.csv` and `knn_three_class_data.csv`. In your written solution, provide a plot of K (x-axis) to classification error (ratio of points misclassified to total; y-axis) for each data set, and report the *best* k for each. Submit your script as a python file named `knn-cv`.

   Solution.

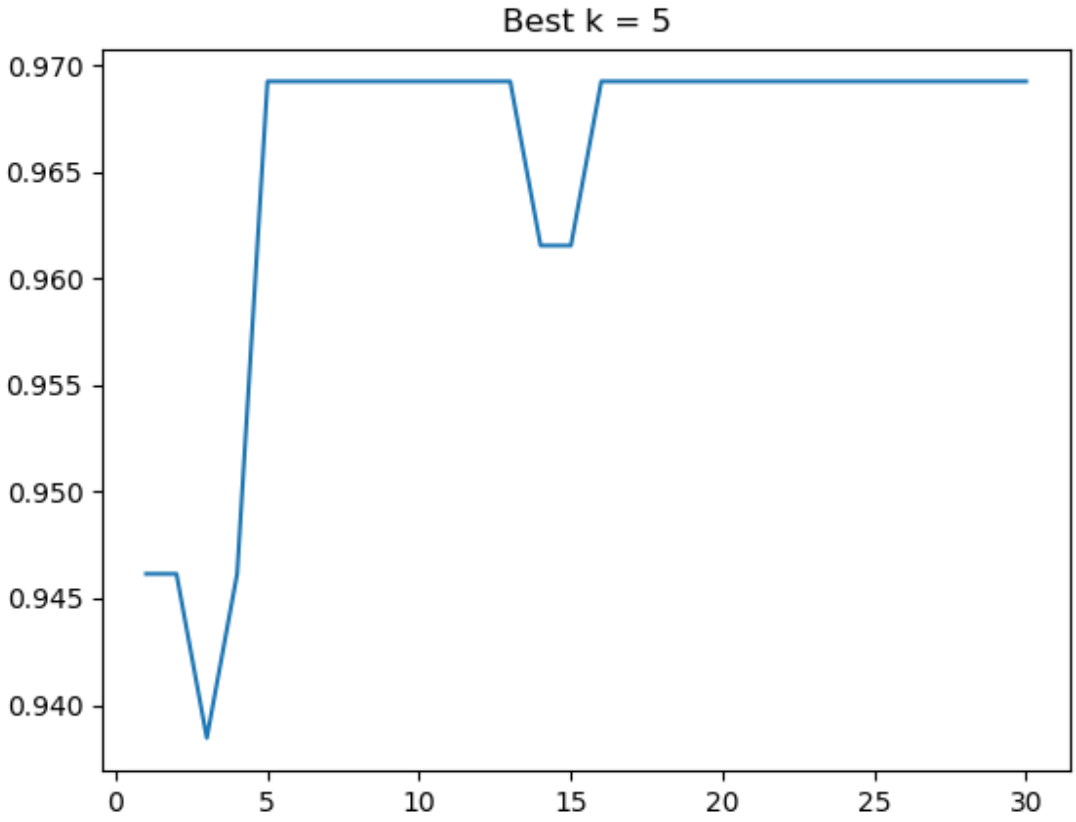

Figure 9: 10-fold cross validation of KNN on binary classification where 1 ¡= k ¡= 30

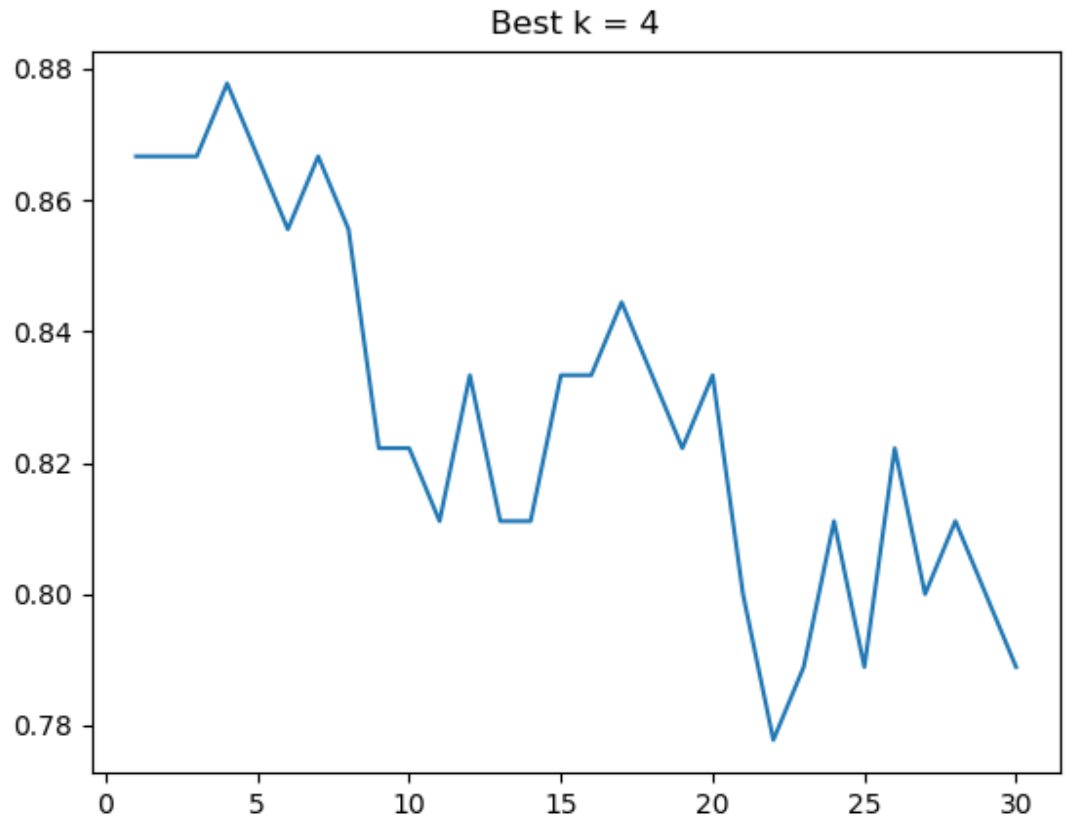

Figure 10: 10-fold cross validation of KNN on three-class classification where 1 ¡= k ¡= 30

```
#!/usr/bin/env python3
"""
Author : kyclark
Date   : 2018-11-24
Purpose: Cross-validation of K-Nearest Neighbors
"""

import argparse
import sys
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from collections import Counter
from scipy.spatial import distance
from sklearn.model_selection import KFold
from sklearn.model_selection import ShuffleSplit
from sklearn.model_selection import train_test_split
```

```
# --------------------------------------------------
def get_args():
    """get command-line arguments"""
    parser = argparse.ArgumentParser(
        description='Cross-validation of K-Nearest Neighbors',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument(
        '-f',
        '--file',
        help='Input file',
        metavar='FILE',
        type=str,
        default='../data/knn_binary_data.csv')

    parser.add_argument(
        '-i',
        '--iterations',
        help='How many iterartions for CV',
        metavar='int',
        type=int,
        default=10)

    parser.add_argument(
        '-o',
        '--outfile',
        help='File name to write figure',
        metavar='FILE',
        type=str,
        default='')

    parser.add_argument(
        '-k', help='Max value for K', metavar='int', type=int, default=30)

    parser.add_argument(
        '-q', '--quiet', help='Do not show figures', action='store_true')

    return parser.parse_args()


# --------------------------------------------------
def warn(msg):
    """Print a message to STDERR"""
    print(msg, file=sys.stderr)


# --------------------------------------------------
def die(msg='Something bad happened'):
    """warn() and exit with error"""
    warn(msg)
    sys.exit(1)
```

```
# --------------------------------------------------
def knn(p, k, x, t):
    """
    K-Nearest Neighbors classifier.  Return the most frequent class among the k
    nearest points
    :param p: point to classify (assumes 2-dimensional)
    :param k: number of nearest neighbors
    :param x: array of observed 2-dimensional points
    :param t: array of target labels (corresponding to points)
    :return: the top class label
    """

    d = np.argsort(list(map(lambda z: distance.euclidean(p, z), x)))[:k]
    count = Counter(t[d])
    return count.most_common(1)[0][0]


# --------------------------------------------------
def main():
    """Make a jazz noise here"""
    args = get_args()
    infile = args.file
    K = args.k
    iterations = args.iterations
    out_file = args.outfile

    df = pd.read_csv(infile)
    X = df.iloc[:, 0:2].values
    t = pd.to_numeric(df.iloc[:, 2].values, downcast='integer')

    kf = KFold(n_splits=iterations)

    x_plot = []
    y_plot = []
    for k in range(1, K + 1):
        predictions = []
        for train, test in kf.split(X):
            X_train, X_test, y_train, y_test = X[train], X[test], t[train], t[
                test]
            predicted = list(
                map(lambda p: knn(p, k, X_train, y_train), X_test))
            predictions.append(np.mean(predicted == y_test))

        print('k {} = {}'.format(k, np.mean(predictions)))
        x_plot.append(k)
        y_plot.append(np.mean(predictions))

    best = np.argmax(y_plot) + 1
```

```
    plt.plot(x_plot, y_plot)
    plt.title('Best k = {}'.format(best))

    if out_file:
        warn('Saving figure to "{}"'.format(out_file))
        plt.savefig(out_file)

    if not args.quiet:
        warn('Showing figure')
        plt.show()

    warn('Done')

# --------------------------------------------------
if __name__ == '__main__':
    main()
```