Homework 11: Classification, Two-Sample Tests, and Tables Review

Please complete this notebook by filling in the cells provided. When you're done, follow the instructions in this short explainer video (https://www.youtube.com/watch?v=gMt_Rq43y_4&ab_channel=FahadKamran) to submit your homework.

If you cannot submit online, come to office hours for assistance. The office hours schedule appears on data8.org/fa16/weekly.html (http://data8.org/fa16/weekly.html).

This assignment is due **Monday, November 28 at 7PM**. You will receive an early submission bonus point if you turn it in by **Wednesday, November 23 at 7PM**. (**Note the unusual dates!**) Directly sharing answers is not okay, but discussing problems with course staff or with other students is encouraged.

Important note: Only Parts 1 and 2 of this assignment will be graded. Parts 3 (a review of table methods you'll need to use in Project 3) and 4 (on overfitting in classification and regression) will not be graded.

Note of optimism: This is the last graded homework assignment of the semester. You're almost done!

Reading:

- Textbook chapter 15 (https://www.inferentialthinking.com/chapters/15/classification.html)
- Textbook chapter 16 (https://www.inferentialthinking.com/chapters/16/comparing-two-samples.html)

Run the cell below to prepare the notebook.

```
In [ ]: # Run this cell to set up the notebook, but please don't change it.
        import numpy as np
        from datascience import *

        # These lines do some fancy plotting magic.
        import matplotlib
        %matplotlib inline
        import matplotlib.pyplot as plt
        plt.style.use('fivethirtyeight')
        import warnings
        warnings.simplefilter('ignore', FutureWarning)
        from matplotlib import patches
        from ipywidgets import interact, interactive, fixed
        import ipywidgets as widgets

        from client.api.assignment import load_assignment
        tests = load_assignment('hw11.ok')
```

1. Reading Sign Language with Classification

Brazilian Sign Language is a visual language used primarily by Brazilians who are deaf. It is more commonly called Libras. People who communicate with visual language are called *signers*. Here is a video of someone signing in Libras:

```
In [ ]: from IPython.lib.display import YouTubeVideo
        YouTubeVideo("mhIcuMZmyWM")
```

Programs like Siri or Google Now begin the process of understanding human speech by classifying short clips of raw sound into basic categories called *phones*. For example, the recorded sound of someone saying the word "robot" might be broken down into several phones: "rrr", "oh", "buh", "aah", and "tuh". Phones are then grouped together into further categories like words ("robot") and sentences ("I, for one, welcome our new robot overlords") that carry more meaning.

A visual language like Libras has an analogous structure. Instead of phones, each word is made up of several *hand movements*. As a first step in interpreting Libras, we can break down a video clip into small segments, each containing a single hand movement. The task is then to figure out what hand movement each segment represents.

We can do that with classification!

The data (https://archive.ics.uci.edu/ml/machine-learning-databases/libras/movement_libras.names) in this exercise come from Dias, Peres, and Biscaro, researchers at the University of Sao Paulo in Brazil. They identified 15 distinct hand movements in Libras (probably an oversimplification, but a useful one) and captured short videos of signers making those hand movements. (You can read more about their work here (http://ieeexplore.ieee.org/Xplore/login.jsp?url=http%3A%2F%2Fieeexplore.ieee.org%2Fiel5%2F5161636%2F5178557%2F05178917.pdf&authDecision=-2 The paper is gated, so you will need to use your institution's Wi-Fi or VPN to access it.)

For each video, they chose 45 still frames from the video and identified the location (in horizontal and vertical coordinates) of the signer's hand in each frame. Since there are two coordinates for each frame, this gives us a total of 90 numbers summarizing how a hand moved in each video. Those 90 numbers will be our *attributes*.

Each video is *labeled* with the kind of hand movement the signer was making in it. Each label is one of 15 strings like "horizontal swing" or "vertical zigzag".

For simplicity, we're going to focus on distinguishing between just two kinds of movements: "horizontal straight-line" and "vertical straight-line". We took the Sao Paulo researchers' original dataset, which was quite small, and used some simple techniques to create a much larger synthetic dataset.

These data are in the file `movements.csv`. Run the next cell to load it.

```
In [6]: movements = Table.read_table("movements.csv")
        movements.take(np.arange(5))
```

The cell below displays movements graphically. Run it and use the slider to answer the next question.

In [7]:
```
# Just run this cell and use the slider it produces.
def display_whole_movement(row_idx):
    num_frames = int((movements.num_columns-1)/2)
    row = np.array(movements.drop("Movement type").row(row_idx))
    xs = row[np.arange(0, 2*num_frames, 2)]
    ys = row[np.arange(1, 2*num_frames, 2)]
    plt.figure(figsize=(5,5))
    plt.plot(xs, ys, c="gold")
    plt.xlabel("x")
    plt.ylabel("y")
    plt.xlim(-.5, 1.5)
    plt.ylim(-.5, 1.5)
    plt.gca().set_aspect('equal', adjustable='box')

def display_hand(example, frame, display_truth):
        time_idx = frame-1
        display_whole_movement(example)
        x = movements.column(2*time_idx).item(example)
        y = movements.column(2*time_idx+1).item(example)
        plt.annotate(
            "frame {:d}".format(frame),
            xy=(x, y), xytext=(-20, 20),
            textcoords = 'offset points', ha = 'right', va = 'bottom',
            color='white',
            bbox = {'boxstyle': 'round,pad=0.5', 'fc': 'black',
'alpha':.4},
            arrowprops = {'arrowstyle': '->', 'connectionstyle':'arc3,ra
d=0', 'color': 'black'})
        plt.scatter(x, y, c="black", zorder=10)
        plt.title("Hand positions for movement {:d}{}".format(example,
"\n(True class: {})".format(movements.column("Movement type").item(examp
le)) if display_truth else ""))

def animate_movement():
    interact(
        display_hand,
        example=widgets.BoundedIntText(min=0, max=movements.num_rows-1,
value=0, msg_throttle=1),
        frame=widgets.IntSlider(min=1, max=int((movements.num_columns-
1)/2), step=1, value=1, msg_throttle=1),
        display_truth=fixed(False))

animate_movement()
```

Question 1

Before we move on, check your understanding of the dataset. Judging by the plot, is the first movement example a vertical motion, or a horizontal motion, or is it difficult to tell? This is the kind of question a classifier has to answer. Find out the right answer by looking at the `"Movement type"` column. (It's okay if you guessed wrong for this one.)

Write your answer here, replacing this text.

Splitting the dataset

We'll do 2 different kinds of things with the `movements` dataset:

1. We'll build a classifier that uses the movements with known labels as examples to classify similar movements. This is called *training*.
2. We'll evaluate or *test* the accuracy of the classifier we build.

For reasons discussed in lecture and the textbook, we want to use separate datasets for these two purposes. So we split up our one dataset into two.

Question 2

Create a table called `train_movements` and another table called `test_movements`. `train_movements` should include the first $\frac{11}{16}$ of the rows in `movements` (rounded to the nearest integer), and `test_movements` should include the remaining $\frac{5}{16}$.

Hint: Use the table method `take`.

In [9]:
```
training_proportion = 11/16
num_movements = movements.num_rows
num_train = int(round(num_movements * training_proportion))

train_movements = ...
test_movements = ...

print("Training set:\t",   train_movements.num_rows, "examples")
print("Test set:\t",       test_movements.num_rows, "examples")
```

In [27]:
```
_ = tests.grade('q1_2')
```

Using only 2 features

First let's see how well we can distinguish two movements (a vertical line and a horizontal line) using the hand position from just a single frame (without the other 44).

Question 3

Make a table called `train_two_features` with only 3 columns: the x and y coordinates for the first frame, and the movement type; and only the examples in `train_movements`.

```
In [11]: train_two_features = ...
         train_two_features
```

```
In [12]: _ = tests.grade('q1_3')
```

Now we want to make a scatter plot of the frame coordinates, where the dots for horizontal straight-line movements have one color and the dots for vertical straight-line movements have another color. Here is a scatter plot without colors:

```
In [13]: train_two_features.scatter("Frame 1 x", "Frame 1 y")
```

This isn't useful because we don't know which dots are which movement type. We need to tell Python how to color the dots. Let's use gold for vertical and blue for horizontal movements.

`scatter` takes an extra argument called `colors` that's the name of an extra column in the table that contains colors (strings like "red" or "orange") for each row. So we need to create a table like this:

Frame 1 x	Frame 1 y	Movement type	Color
0.522768	0.769731	vertical straight-line	gold
0.179546	0.658986	horizontal straight-line	blue
...	...	...	...

Question 4

In the cell below, create a table named `with_colors`. It should have the same columns as the example table above, but with a row for each row in `train_two_features`.

```
In [14]: # You should find the following table useful.
         type_to_color = Table().with_columns(
             "Movement type", make_array("vertical straight-line", "horizontal st
         raight-line"),
             "Color",         make_array("gold",                   "blue"))

         with_colors = ...
         with_colors.scatter("Frame 1 x", "Frame 1 y", colors="Color")
```

```
In [15]: _ = tests.grade('q1_4')
```

Question 5

Based on the scatter plot, how well will a nearest-neighbor classifier based on only these 2 features (the x- and y-coordinates of the hand position in the first frame) work? Will it:

1. distinguish almost perfectly between vertical and horizontal movements;
2. distinguish somewhat well between vertical and horizontal movements, getting some correct but missing a substantial proportion; or
3. be basically useless in distinguishing between vertical and horizontal movements?

Why?

Write your answer here, replacing this text.

Question 6

Imagine training a 1-nearest-neighbor classifier on this data. For an example with first-frame coordinates (0.8, 0.9), what are the (rough) coordinates of the nearest point in the training set, and what is its class?

Write your answer here, replacing this text.

A classifier

Now let's do nearest-neighbor classification. In the interest of time, we'll do some of the steps for you. Your job will be to *read and understand the given code* and use it to complete the classifier.

In the following cell, we've defined three functions. They're documented in the cell itself. Unless you want to write your own classifier from scratch, use these to help you answer the next question.

In [16]:
```
def euclidean_distance(a, b):
    """
    The Euclidean distance (ordinary straight-line distance) between
    the numbers in arrays a and b.

    You don't need to use this function; it's designed to be used
    by the k_nearest_rows function below.
    """
    return sum((a - b)**2)**0.5

def k_nearest_rows(attributes, data, k):
    """
    The k nearest neighbors in the data table to an observation with the
 given attributes.

    This function takes 3 arguments:
      * attributes: An array of numbers, each one an attribute of the
        observation whose neighbors we're finding.
      * data: A table with many rows.  The columns in this table are
        the attributes of the data (the same as the elements of the
        attributes array), except that the last column is a label,
        not a feature.  So data.num_columns should be 1 bigger than
        len(attributes).
      * k: The number of neighbors to find.

    It returns a table containing the k rows of data whose attributes
    are closest in Euclidean distance to the attributes array.  The
    returned table has the same columns as the data table.
    """
    distances = make_array()
    data_without_labels = data.drop(data.num_columns-1)
    for i in np.arange(data.num_rows):
        training_example_attributes =
np.array(data_without_labels.row(i))
        distance = euclidean_distance(attributes, training_example_attri
butes)
        distances = np.append(distances, distance)

    return data.with_column("distance to observation", distances)\
               .sort("distance to observation")\
               .take(np.arange(k))\
               .drop("distance to observation")

def most_common(things):
    """
    The most common element in an array of anything.

    This function takes 1 argument:
      * things: An array containing any kind of data.

    It returns the most common element of that array.  For example,
      most_common(make_array(0, 1, 1, 2, 1))
    is 1, and
      most_common(make_array("the", "I", "a", "I"))
    is "I".  If there's a tie, one of the most common elements is
    chosen arbitrarily.  If the array is empty, an error will
    happen.
    """
    counts = Table().with_column("things", things).group("things")
    return counts.sort("count", descending=True).column(0).item(0)
```

Question 7

Below is the skeleton of a function that classifies examples as either vertical straight-line movements or horizontal straight-line movements. Fill it in so that it does what its documentation promises.

Hint: Use the functions we've defined for you.

In [17]:
```
# Fill in this function.
def classify(attributes, training_data, k):
    """
    Classifies an observation with the given attributes.

    This function takes three arguments:
     * attributes: An array of numbers.  Each is an attribute of a
       movement.
     * training_data: A table of observations.  Must have one column
       for each attribute in the attributes array, *plus* one column
       at the end containing the true class of each observation.
     * k: The number of neighbors to look at.

    It returns a string, which is the name of the movement
    type we guess for the given movement.

    We compute this classification by k-nearest-neighbors.  Our
    training data (the data in which we look for neighbors) is the
    given training_data table.
    """
    ...
```

Verifying that `classify` doesn't have bugs

Question 8

Verify that `classify` works as expected by replicating the simple example in question 6. That is, classify an hypothetical movement with first-frame coordinates (0.8, 0.9), using a 1-nearest neighbor classifier with `train_two_features` as the training set. Give the name `example_1NN_classification` to the predicted class of the observation. Make sure it matches your answer to question 6!

In [18]:
```
example_1NN_classification = ...
example_1NN_classification
```

Question 9

Using the scatter plot you made in question 4, find a movement that would be classified as horizontal by a ***3***-nearest neighbor classifier, again with `train_two_features` as the training set. Verify that `classify` produces the expected classification for that movement.

In [19]:
```
example_3NN_classification = ...
example_3NN_classification
```

Testing the accuracy

Let us compute the accuracy of our classifier.

First, we'll continue to imagine we have only 2 features rather than the full 90. This will verify your answer to question 5.

The next cell defines a function called `classify_all`. It will be useful in answering the next question, so read its documentation.

In [20]:
```
def classify_all(test_data, training_data, k):
    """
    Classifies each observation in test_data, using k-nearest neighbors
 with training_data.

    This function takes three arguments:
     * test_data: A table of observations.  Each row contains the
       attributes of one movement.  There should be no extra columns;
       in particular, this table shouldn't include a column with the
       true classes.
     * training_data: A table of observations.  Must have one column
       for each attribute in the test_data table, *plus* one column
       at the end containing the true class of each observation.
     * k: The number of neighbors to look at.

    It returns an array of strings, the classification of each row
    in test_data.

    We compute these classification by k-nearest-neighbors.  Our
    training data (the data in which we look for neighbors) is the
    given training_data table.
    """
    classifications = make_array()
    for i in np.arange(test_data.num_rows):
        classification = classify(
            np.array(test_data.row(i)),
            training_data,
            k)
        classifications = np.append(classifications, classification)
    return classifications
```

Question 10

Create a table called `test_two_features` containing the observations in `test_movements` with only the first 2 features. This table should have the same columns as `train_two_features`. Compute the classifications of those observations using a 3-nearest neighbor classifier with `train_two_features` as the training set. Then compute the proportion of those observations that are correct.

Note: This might take around a minute.

```
In [21]: test_two_features = ...
         two_features_classifications = ...
         two_features_proportion_correct = ...
         two_features_proportion_correct
```

Does that match what you wrote in question 5?

Now let's try out the classifier with all 90 features. We'd expect this to work better, because we're giving the classifier more information about each example. (We've seen that some horizontal movements and vertical movements start out in the same place, but hopefully by the last frame those movements end up in very different places.)

Question 11

Compute the accuracy of a 3-nearest neighbor classifier using `train_movements` as the training set and `test_movements` as the test set, *when all 90 features are used*.

Note: This will take a few minutes.

```
In [22]: all_features_classifications = ...
         proportion_correct = ...
         proportion_correct
```

If you're interested, the next cell displays the movements in the test set that your classifier got wrong. Would you have done better yourself?

```
In [23]: wrong_examples = np.flatnonzero(all_features_classifications != test_mov
         ements.column("Movement type"))
         interact(
                 display_hand,
                 example={str(i+num_train): i+num_train for i in wrong_examples},
                 frame=widgets.IntSlider(min=1, max=int((movements.num_columns-
         1)/2), step=1, value=1, msg_throttle=1),
                 display_truth=fixed(True));
```

Question 12

Would you predict that the accuracy would go up or down (relative to the result in question 11) if we used `train_movements` as both the training set and the test set? Why? You can try it yourself to verify your answer.

Write your answer here, replacing this text.

You're done! If you want more practice, try rewriting the `most_common` or `k_nearest_rows` functions for yourself.

2. Testing Relationship Ratings

Throughout this part of the homework, we will be interested in learning how different factors can contribute to the rating couples give their relationship, on a scale of 1-5 (1 being most satisfied, and 5 being the least).

To do this, we will be looking at the table that is loaded below.

Note: The data in this table is quite dated. It only includes heterosexual relationships. A better dataset would contain a more accurate sample of couples. When drawing conclusions from data, it is important to be aware of biases that exist in the dataset.

In [3]:
```
couples = Table.read_table('couples.csv')
couples
```

Looking at the table above, we note that we should be able to test whether different factors can affect the relationship rating, from age to education. To begin, we are first going to investigate whether marital status has an effect on relationship rating. In order to accomplish this, we are going to do a permutation test!

Question 1

State the null and alternative hypothesis for such a test (assume we aren't trying to figure out ***how*** marital status affects relationship rating, just whether or not it does).

Answer:

Null hypothesis:

Alternate hypothesis:

Question 2

Before we continue, we will have to talk about the notion of a contingency table.

Define `contingency` to be a table where the rows represent unique values of the relationship ratings, the columns represent the unique values of marital status present in the table, and the values in each cell represent the number of individuals with a given marital status and relationship rating.

Hint: Use `pivot`.

```
In [4]: contingency = ...
        contingency
```

```
In [5]: _ = tests.grade('q2_2')
```

Question 3

It is difficult to identify a patterin in the above table, since our table has so many more married people than it does unmarried. To rectify this, we will instead compute proportions.

Assign `contingency_prop` to a table with the same columns and rows, but the values are instead in proportions, i.e. the value in the cell corresponding to a relationship rating of 1 and marital status of "married" should be the proportion of married people who gave their relationship a 1.

```
In [6]: #The staff solution took 5 lines
        contingency_prop = ...
```

Question 4

Assign `tvd_marriages` to the total variational distance between the distributions of married people's relationship ratings and unmarried people's relationship ratings.

```
In [8]: tvd_marriages = ...
        tvd_marriages
```

```
In [9]: _ = tests.grade('q2_4')
```

Question 5

Let's get more general now. Define `tvd` which takes in

- any table like `couples`
- `conditions`: the name of a column like Relationship Rating
- `values`: the name of a column like Marital Status

The function should return the total variation distance between the distribution of conditions in the first unique value of values and the distribution of conditions in the second unique value of values (assume there are only two unique values)

Hint: This looks daunting, but you are just generalizing your code from above.

```
In [17]: def tvd(t, conditions, values):
             return ...

         tvd(couples, 'Relationship Rating', 'Marital Status')
```

Question 6

Now that we have a valid test statistic, fill in the implementation of `permutation_test` below. It should run a permutation test to check whether conditions and values are related, where conditions and values are defined as above. See section 16.1 (https://www.inferentialthinking.com/chapters/16/1/two-categorical-distributions.html) for details about the permutation test.

The inputs to the function are:

- `t`: a table containing our data
- `conditions`: the name of the column in `t` representing our conditions
- `values`: the name of the column in `t` representing our conditions
- `repetitions`: the number of permutations we'd like to test
- `statistic`: a test statistic function (such as `tvd`).

The return value should be the p-value of seeing the observed value assuming the null hypothesis. Fill in the implementation as necessary; we have also provided code which allows you to see an empirical histogram of the distribution under the null.

In [16]:
```
def permutation_test(t, conditions, values, repetitions , statistic):
    stats = ...

    for i in np.arange(repetitions):
        shuffled = ...
        shuffled_table = Table().with_columns(conditions, t.column(condi
tions), values, shuffled.column(values))
        stats = ...

    observation = ...
    p_value = ...

    Table().with_column('Empirical distribution of TVD', stats).hist(bin
s=np.arange(-.02, .12, .02))
    return p_value

permutation_test(couples, 'Relationship Rating', 'Marital Status', 200,
tvd)
```

Question 7

Based on the above method, should we reject the null hypothesis that the relationship ratings and marital status areun correlated, or should we fail to reject? Why?

Write your answer here, replacing this text.

3. Review of Table Methods

In this question, we are going to be going over some important table concepts that might be useful for the project. The dataset we will be using is `twitter_follows.csv`, which you can load below.

```
In [3]: twitter_follows = Table.read_table('twitter_follows.csv')
        twitter_follows
```

Question 1

Assign `least_followed` to the screename of the person who has the fewest followers. Do this using code; do not just inspect the table.

```
In [6]: least_followed = ...
        least_followed
```

Question 2

Assign `most_followed` to the screename of the person who has the most followers now.

```
In [7]: most_followed = ...
        most_followed
```

Question 3

Assign `avg_friends` to be the average number of friends across all of these users.

```
In [8]: avg_friends = ...
        avg_friends
```

Question 4

Define a function `comparison` which takes in a value and checks whether or not it's above `avg_friends`. If the value is greater than or equal to the average amount of friends, it returns True, otherwise False.

```
In [ ]: def comparison(val):
            ...
        comparison(1000)
```

Question 5

Using apply, assign `avg_comparison` to a table which has an extra column called "Above Average?" which is values of true or false corresponding to whether or not the user has more or less friends than the average. Think about using the function you defined above, you **must** use apply for this question.

```
In [10]: above_avg_array = ...
         avg_comparison = twitter_follows.with_column('Above Average?', above_avg
         _array)
         avg_comparison
```

4. The Gravity of Overfitting

Imagine you are an early natural scientist trying to understand how fast things fall. You run experiments in which you drop an iron ball from a very tall cliff. You want to know the relationship between the length of time (in seconds) the ball has been dropping and the distance the ball has fallen (in meters). Isaac Newton will later predict, using calculus and a model of physics, that the relationship is

$$\text{distance fallen} = 0 + 0 \times (\text{time spent falling}) + \frac{g}{2} \times (\text{time spent falling})^2,$$

where g is a constant related to the gravity of Earth. But that hasn't happened yet.

You drop your iron ball 7 times from the cliff. Each time, you choose a time between 0 and 3 seconds (uniformly at random) and measure how far the ball has fallen after that long. Your distance measurements rely on a human assistant with a stopwatch standing on the ground, so they are somewhat noisy. That is, their measurements of the distance fallen aren't exactly equal to the actual distance fallen, but they're kinda close.

You have three hypotheses:

1. Distance is a linear function of falling time. That means $\text{distance fallen} = b + a \times (\text{time spent falling})$ for some numbers a and b.
2. Distance is a quadratic function of falling time. That means $\text{distance fallen} = c + b \times (\text{time spent falling}) + a \times (\text{time spent falling})^2$ for some numbers a, b, and c.
3. Distance is a 6th-degree ("hexic") polynomial function of falling time. (A 6th-degree polynomial has 7 parameters and can "wiggle around" a lot more than a linear or quadratic function. You don't need to know anything else about polynomials to work on this question.)

To test your hypotheses, you decide to find the function that fits the data most closely under each hypothesis - that is, the linear, quadratic, or 6th-degree polynomial function with the smallest mean squared error for these 7 points.

The cell below loads the data, finds the best-fitting linear, quadratic, and 6th-degree polynomial functions for the dataset, and plots them. (You don't need to know what it's doing, but the part that calls `minimize` should look somewhat familiar.)

```
In [2]: falls = Table.read_table("falls.csv")

        def polynomial(coefficients, x):
            """The value of a polynomial with the given coefficients, for input
         x."""
            num_coefficients = len(coefficients)
            powers_of_x = x**np.arange(num_coefficients)
            return sum(coefficients * powers_of_x)

        def polynomial_predictions(coefficients, x_values):
            """The values of a polynomial with the given coefficients, for each
         value in the array x_values."""
            def prediction(x):
                return polynomial(coefficients, x)
            return Table().with_columns("x", x_values).apply(prediction, "x")

        def mse(coefficients):
            """The mean squared error when we use a polynomial with the given co
        efficients to predict distance fallen."""
            predictions = polynomial_predictions(coefficients, falls.column("fal
        ling time (s)"))
            return np.mean((predictions - falls.column("distance fallen
        (m)"))**2)

        def print_polynomial(coefficients):
            xs = ["*x^" + str(power) if power > 1 else "*x" if power == 1 else
        "" for power in np.arange(len(coefficients))]
            print("Best-fit degree-{:d}
        polynomial:".format(len(coefficients)-1),
                  " + ".join(["{0: >#06.4g}".format(c) + x for c, x in zip(coeff
        icients, xs)]))

        def plot_polynomial(coefficients):
            """Plot the data, plus a polynomial with the given coefficients."""
            falling_times = np.arange(0, 3, .01)
            predictions = polynomial_predictions(coefficients, falling_times)
            falls.scatter(0, zorder=10, label="observations")
            plt.plot(falling_times, predictions, label="best-fit degree-{:d} pol
        ynomial".format(len(coefficients)-1))
            plt.legend(bbox_to_anchor=make_array(2, .6))
            plt.ylim(-10, 50)

        for num_coefficients in make_array(2, 3, 7):
            best_polynomial = minimize(mse, start=np.zeros(num_coefficients), ar
        ray=True, method='SLSQP', options={'ftol': 1e-10})
            print_polynomial(best_polynomial)
            plot_polynomial(best_polynomial)
```

Note: The 6th-degree polynomial doesn't actually have a discontinuity between 2.1 and 2.7; it just varies so sharply that we couldn't fit the whole curve on the same scale as the linear and quadratic curves without making those curves look very flat.)

Question 1

By examining the three plots, rank the curves by mean squared residual on the 7 data points, least to greatest. If you think there is a tie between any of the curves, say that.

Write your answer here, replacing this text.

You aren't convinced by the results of this experiment, so you decide to run a second copy. You drop 7 iron balls for random amounts of time between 0 and 3 seconds and record the distance fallen. Then you check how accurately the three functions computed above (the same 3 curves pictured in the plots) predicted the distance fallen for these 7 new trials. You compute the MSE for each one.

Question 2

Rank the 3 curves by the mean squared error you'd *expect* to see for these 7 new trials, least to greatest. If you think there is a tie between any of the curves, say that.

Write your answer here, replacing this text.

The problem with the 6th-degree polynomial is that it's able to "wiggle around" too much to fit the data we train it on. 1-nearest neighbor classifiers have an analogous problem.

The cell below loads a (synthetic) dataset about art forgery. Each row is a painting that is either a forgery or an original painting by the (fictional) Impressionist artist Edgar Detas. Brush strokes are thought to be highly personal and difficult to replicate, so for each painting, we have measured two characteristics: the average width of brush strokes (in millimeters) and the average length of brush strokes (also in millimeters).

Gold dots represent real paintings. Blue dots represent forgeries.

```
In [24]: forgeries_sample = Table.read_table("forgeries_sample.csv")
         forgeries_sample.scatter("Stroke width", "Stroke length",
         colors="Color")

         def set_graph_dimensions():
             # This function just sets the width and height of the graph.
             # We define it here so we can reuse it in the next cell.
             plt.xlim(0, 10)
             plt.ylim(2, 14)
             plt.gca().set_aspect('equal', adjustable='box')

         set_graph_dimensions()
```

The next cell shows the decision regions of a 1-nearest neighbor classifier and a 5-nearest neighbor classifier on this dataset.

In [25]:
```
def make_classifier(k, training):
    """Train a k-NN classifier on the given art training data."""
    attributes = training.select("Stroke width", "Stroke 
length").to_df().as_matrix()
    classes = training.column("Forgery")
    def classify(w, l):
        distances = np.sum((attributes - make_array(w, l))**2, axis=1)
        closest_example_indices = np.argsort(distances)[:k]
        return 2*sum(classes[closest_example_indices]) >= k
    return classify

def display_decision_region(classifier):
    xs = np.arange(.5, 9, .1)
    ys = np.arange(4, 13, .1)
    x_grid, y_grid = np.meshgrid(xs, ys)
    decisions = Table().with_columns(
        "Stroke width", x_grid.flatten(),
        "Stroke length", y_grid.flatten())
    decisions = decisions\
        .with_column("Forgery", decisions.apply(classifier, 
make_array(0, 1)))\
        .join("Forgery", Table().with_columns("Forgery", make_array(0, 
1), "Color", make_array("Gold", "Blue")))\
        .select("Stroke width", "Stroke length", "Color")
    decisions.scatter("Stroke width", "Stroke length", colors="Color", a
lpha=.4, s=2)
    plt.scatter(
        forgeries_sample.column("Stroke width"),
        forgeries_sample.column("Stroke length"),
        c=forgeries_sample.column("Color"),
        lw=1,
        edgecolor="black")
    set_graph_dimensions()

one_nn = make_classifier(1, forgeries_sample)
five_nn = make_classifier(5, forgeries_sample)
display_decision_region(one_nn)
display_decision_region(five_nn)
```

Question 3

Suppose we use each classifier to classify the paintings in `forgery_sample`. (Yes, the training set.) Rank the classifiers by the proportion of **correct** guesses they make, **greatest to least**. (Look at the plots generated by the cells above to answer this question. You can run the classifiers to verify your answer if you want.)

Write your answer here, replacing this text.

The decision region of the 1-nearest neighbor classifier is pocked with little holes. For example, notice the two regions in the bottom-left where the 1-nearest neighbor classifier classifies paintings as forgeries.

Question 4

Suppose we find some **new** paintings. We're not sure whether they are Detas originals or forgeries, so we ask our 2 classifiers. For new paintings that fall in those two bottom-left regions, do the classifiers make different guesses? If so, which classifier do you think makes better guesses?

Write your answer here, replacing this text.

```
In [ ]: # For your convenience, you can run this cell to run all the tests at on
        ce!
        import os
        print("Running all tests...")
        _ = [tests.grade(q[:-3]) for q in os.listdir("tests") if
        q.startswith('q')]
        print("Finished running all tests.")
```