CS/ECE/ME 532
Homework 4: orthogonality and more least squares

due: Friday October 14, 2016

1. **Range-Nullspace duality.** Suppose $A \in \mathbb{R}^{m\times n}$ is any matrix. Further suppose that $p \in \text{range}(A)$ and $q \in \text{null}(A^\mathsf{T})$. Prove that $p^\mathsf{T} q = 0$.

   SOLUTION: If $p \in \text{range}(A)$, that means that $p = Ax$ for some $x \in \mathbb{R}^n$. Therefore:

$$p^\mathsf{T} q = (Ax)^\mathsf{T} q = x^\mathsf{T}(A^\mathsf{T} q)$$

   However, $q \in \text{null}(A^\mathsf{T})$, therefore $A^\mathsf{T} q = 0$, and so $p^\mathsf{T} q = 0$.

2. **Orthogonal columns.** Consider the matrix and vector

$$A = \begin{bmatrix} 3 & 1 \\ 0 & 3 \\ 0 & 4 \end{bmatrix} \quad \text{and} \quad b = \begin{bmatrix} 1 \\ 3 \\ 1 \end{bmatrix}$$

   a) By hand, find two orthonormal vectors that span the plane spanned by columns of A.

   b) Make a sketch of these vectors and the columns of A in three dimensions.

   SOLUTION:

   (a) We will apply Gram-Schmidt to the columns of A. Starting with the first vector:

$$a_1' = \begin{bmatrix} 3 \\ 0 \\ 0 \end{bmatrix}$$

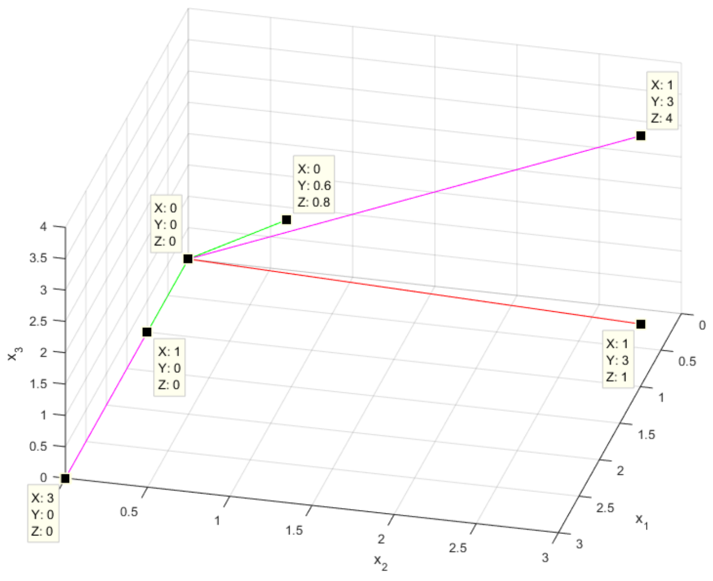
   The next step is to subtract from the second vector its projection onto the first vector:

$$a_2' = a_2 - \frac{a_1'^\mathsf{T} a_2}{a_1'^\mathsf{T} a_1'} a_1' = \begin{bmatrix} 1 \\ 3 \\ 4 \end{bmatrix} - \frac{3+0+0}{9+0+0} \begin{bmatrix} 3 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 3 \\ 4 \end{bmatrix}$$

   Normalizing the two vectors, we obtain:

$$U = \begin{bmatrix} 1 & 0 \\ 0 & \frac{3}{5} \\ 0 & \frac{4}{5} \end{bmatrix}$$

   (b) Here is a sketch. Again, we will accept anything that is roughly correct. A-vectors in magenta, u-vectors in green, b-vector in red.



c) Use these vectors to compute the LS estimate $\widehat{b} = A(A^\top A)^{-1}A^\top b$.

SOLUTION: knowing the orthogonalization of A makes the computation much easier! The least-squares solution $\widehat{b}$ will be the same if we replace A by U. Then, we have:

$$\begin{aligned}
\widehat{b} &= U(U^\top U)^{-1}U^\top b \\
&= UU^\top b \\
&= \begin{bmatrix} 1 & 0 \\ 0 & \frac{3}{5} \\ 0 & \frac{4}{5} \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & \frac{3}{5} \\ 0 & \frac{4}{5} \end{bmatrix}^\top \begin{bmatrix} 1 \\ 3 \\ 1 \end{bmatrix} \\
&= \begin{bmatrix} 1 & 0 \\ 0 & \frac{3}{5} \\ 0 & \frac{4}{5} \end{bmatrix} \begin{bmatrix} 1 \\ \frac{13}{5} \end{bmatrix} \\
&= \begin{bmatrix} 1 \\ \frac{39}{25} \\ \frac{52}{25} \end{bmatrix}
\end{aligned}$$

Where we used the fact that $U^\top U = I$ because U is orthogonal. We can also check that the answer is the same using A (but more work):

$$
\begin{aligned}
\widehat{b} &= A(A^\mathsf{T} A)^{-1} A^\mathsf{T} b \\
&= \begin{bmatrix} 3 & 1 \\ 0 & 3 \\ 0 & 4 \end{bmatrix} \left(\begin{bmatrix} 3 & 1 \\ 0 & 3 \\ 0 & 4 \end{bmatrix}^\mathsf{T} \begin{bmatrix} 3 & 1 \\ 0 & 3 \\ 0 & 4 \end{bmatrix} \right)^{-1} \begin{bmatrix} 3 & 1 \\ 0 & 3 \\ 0 & 4 \end{bmatrix}^\mathsf{T} \begin{bmatrix} 1 \\ 3 \\ 1 \end{bmatrix} \\
&= \begin{bmatrix} 3 & 1 \\ 0 & 3 \\ 0 & 4 \end{bmatrix} \begin{bmatrix} 9 & 3 \\ 3 & 26 \end{bmatrix}^{-1} \begin{bmatrix} 3 \\ 14 \end{bmatrix} \\
&= \frac{1}{225} \begin{bmatrix} 3 & 1 \\ 0 & 3 \\ 0 & 4 \end{bmatrix} \begin{bmatrix} 26 & -3 \\ -3 & 9 \end{bmatrix} \begin{bmatrix} 3 \\ 14 \end{bmatrix} \\
&= \frac{1}{225} \begin{bmatrix} 3 & 1 \\ 0 & 3 \\ 0 & 4 \end{bmatrix} \begin{bmatrix} 36 \\ 117 \end{bmatrix} \\
&= \frac{1}{225} \begin{bmatrix} 225 \\ 351 \\ 468 \end{bmatrix} \\
&= \begin{bmatrix} 1 \\ \frac{39}{25} \\ \frac{52}{25} \end{bmatrix}
\end{aligned}
$$

3. Gram-Schmidt. Write your own code to perform Gram-Schmidt orthogonalization. Your code should take as input a matrix $A \in \mathbb{R}^{m\times n}$ and return as output a matrix $U \in \mathbb{R}^{m\times r}$ where U is orthogonal and has the same range as A. Note that r will indicate the rank of A, so your code can also be used to find the rank of a matrix!

SOLUTION: Here is a (spectacularly short) solution in Matlab:

```
function U = gram_schmidt( A )
   % GRAM-SCHMIDT CODE
   %  U = gram_schmidt( A )
   %  where U'*U=I and range(U)=range(A)
   %  number of columns of U is the rank of A.

   [m,n] = size(A);
   U = zeros(m,0);           % start with empty matrix
   for i = 1:n
      v = A(:,i);            % the current column of A
      v = v - U*(U'*v);      % project onto current output set
      if norm(v) > 1e-12     % ensure linear independence
         U = [U v/norm(v)]; % normalize and add to the set
      end
   end
end  % end of function
```

a) Test your code by applying it to Problem 2 above.

b) Use your code to determine the rank of the following matrices and compare the result to Matlab's `rank` function (or Python's `numpy.linalg.matrix_rank` function, etc.).

$$A_1 = \begin{bmatrix} 3 & 1 & 2 \\ 0 & 3 & 3 \\ 0 & 4 & 4 \\ 6 & 1 & 4 \end{bmatrix} \qquad A_2 = \begin{bmatrix} 1 & 1 & 2 \\ 0 & 3 & 3 \\ 0 & 4 & 4 \\ 3 & 1 & 4 \end{bmatrix}$$

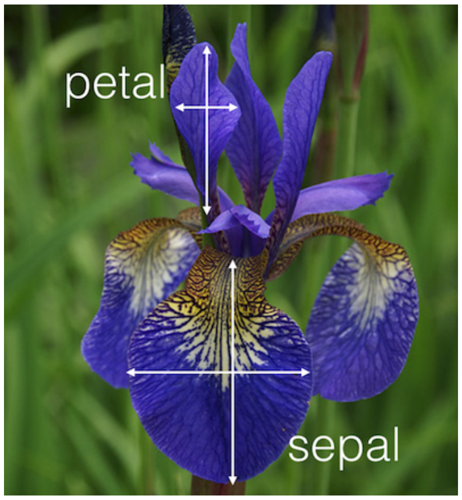
SOLUTION:

(a) Result is the same as the solution computed to problem 1, as expected.

(b) Applying the code to A_1 results in a 4×3 matrix (rank 3). Applying the code to A_2 results in a 4×2 matrix (rank 2)

(see next page for problem 4)

4. Fisher's Iris Classification. In 1936 Ronald Fisher published a famous paper on classification titled "The use of multiple measurements in taxonomic problems." In the paper, Fisher study the problem of classifying iris flowers based on measurements of the sepal and petal widths and lengths, depicted in the image below.



Fisher's dataset is available in Matlab (`fisheriris.mat`) and is widely available on the web (e.g., Wikipedia). The dataset consists of 50 examples of three types of iris flowers. The sepal and petal measurements can be used to classify the examples into the three types of flowers.

a) Formulate the classification task as a least squares problem. For the labels, use setsoa $= -1$, versicolor $= 0$, virginica $= 1$. Then, classify based on which of the labels $\widehat{b}$ is closest to. We'll define the *average classification error* to be:

$$\frac{\text{number of misclassified examples}}{\text{total number of classified examples}}$$

What is the average classification error when your classifier is used on the entire data set?

SOLUTION: Here is Matlab code that computes the average classification error

```
load fisheriris
A = meas;
b = [-ones(50,1); zeros(50,1); ones(50,1)];

bhat = A*(A\b); % least squares predictions

for i = 1:150
    if  abs(bhat(i)) < 0.5  % versicolor
        bhat(i) = 0;
    else
        bhat(i) = sign(bhat(i)); % setsoa or virginica
    end
end
err = mean(bhat~=b)  % average classification error
```

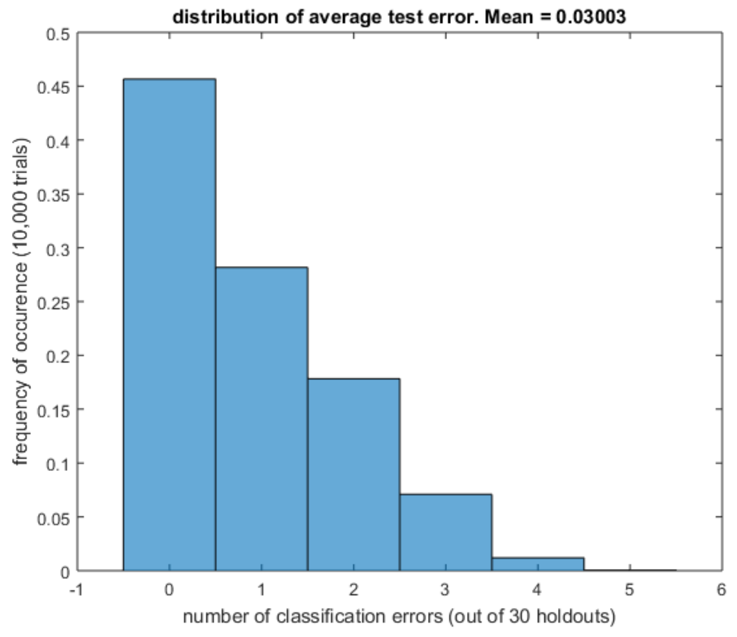
The average classification error turns out to be 2.67%.

b) Let's use cross-validation this time. Write code to train a LS classifier based on 40 labeled examples of each of the three flower types, and then test the performance (by computing the average classification error) on the remaining 10 examples from each type. Repeat this with 10,000 different randomly chosen subsets of training and test. What is the average of all classification errors computed?

SOLUTION: The code below finds a random training set of size 40 (for each label), computes the classifier (using the method of part a), computes the error on the holdout set of 10 (for each label), then finds the average classification error. This is repeated 10,000 times and the average classification error is recorded. I also plotted the error distribution. The holdout set has 30 elements, and random classifiers have 0/30 error roughly 45% of the time and 1/30 error roughly 27% of the time. Classifiers that make 4/30 or more errors are very rare. Here is the code:

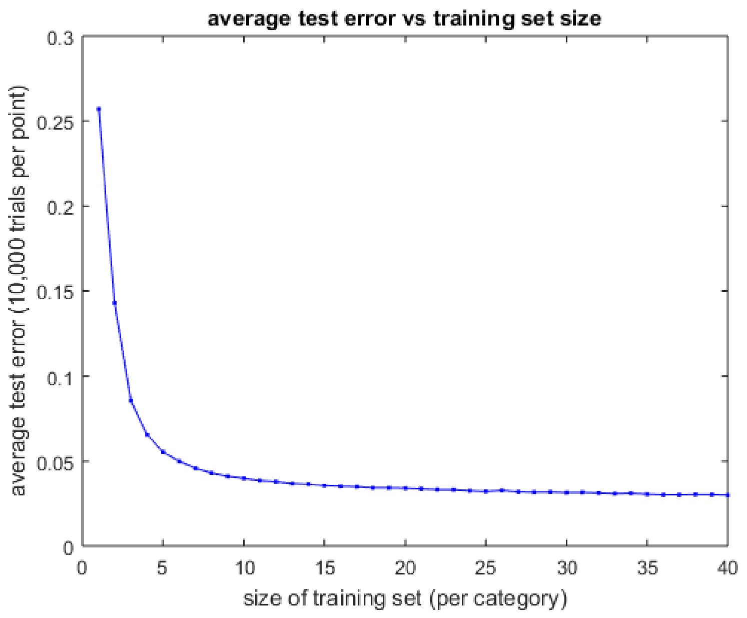
```
load fisheriris
A = meas;
b = [-ones(50,1); zeros(50,1); ones(50,1)];

N = 10000;              % number of random trials
errs = zeros(N,1);      % where we store error values
num_train = 40;         % size of training set

for i = 1:N

   % randomly pick training and holdout sets
   r = randperm(50);
   r = r(1:num_train);             % random set for training
   rc = setdiff(1:50,r);           % remaining are the holdouts
   train = [r r+50 r+100];         % training set
   holdout = [rc rc+50 rc+100];    % holdout set

   % train the classifier
   At = A(train,:);
   bt = b(train,:);
   xt = At\bt;

   % use classifier on holdout set
   Ah = A(holdout,:);
   bh = b(holdout,:);
   bhat = Ah*xt;

   % apply rounding to find labels
   nh = numel(holdout);
   for j = 1:nh
      if abs(bhat(j)) < 0.5
         bhat(j) = 0;
      else
         bhat(j) = sign(bhat(j));
      end
   end
   errs(i) = mean(bhat~=bh);
end

% plot histogram and mean value
avg_error = mean(errs)
figure(1); clf
histogram(errs*nh,'BinMethod','integers','Normalization','probability')
xlabel('number of classification error (out of 30 holdouts)')
ylabel('frequency of occurence (10,000 trials)')
title(['distribution of average classification error. Mean = ' num2str(avg_error)
```

and here is a plot of the histogram produced by the code. The average classification error over 10,000 trials was 3%.



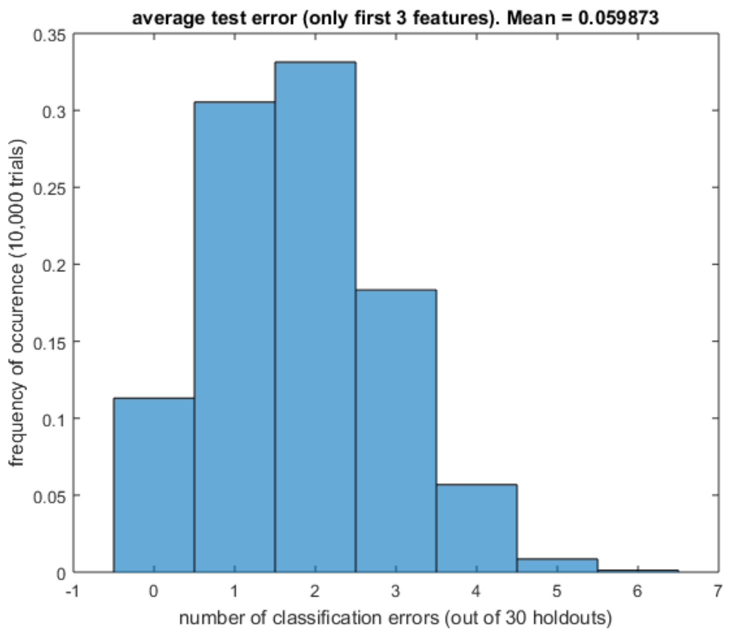
c) Experiment with even smaller sized training sets. Clearly we need at least one training example from each type of flower. Make a plot of average classification error as a function of training set size.

SOLUTION: The code for this part is very similar to part (b) so we omit it. When we reduce the size of the training set, we find that the average error remains relatively constant down to about 10 samples per category (flower). This amounts to roughly 20% of the total data. Any less, and the error goes up dramatically. As we might expect, more training data means a better classifier!



d) Now design a classifier using only the first three measurements (sepal length, sepal width, and petal length). What is the average classification error in this case?

SOLUTION: By slightly modifying the code of part (b), (changing `meas` to `meas(:,1:3)`) we can compute the average error that results from using only the first three features. The new histogram is shown below. This time, the error is about 5.9%.



e) Use a 3d scatter plot to visualize the measurements in (d). Can you find a 2-dimensional subspace that the data approximately lie in? You can do this by rotating the plot and looking for plane that approximately contains the data points.

SOLUTION: Upon making the scatter plot (shown below), it is clear that the points corresponding to different species are clustered (we will see how to address clustering directly later in the class!). It is also clear that there is a lower-dimensional subspace that roughly contains the data. In this case, it is a 2D hyperplane.

To estimate the orientation of this plane (and compute a projection), we need to find a basis that spans the subspace. One approximate way to do this is to pick three representative points (see labels on the figure). The points I chose were:

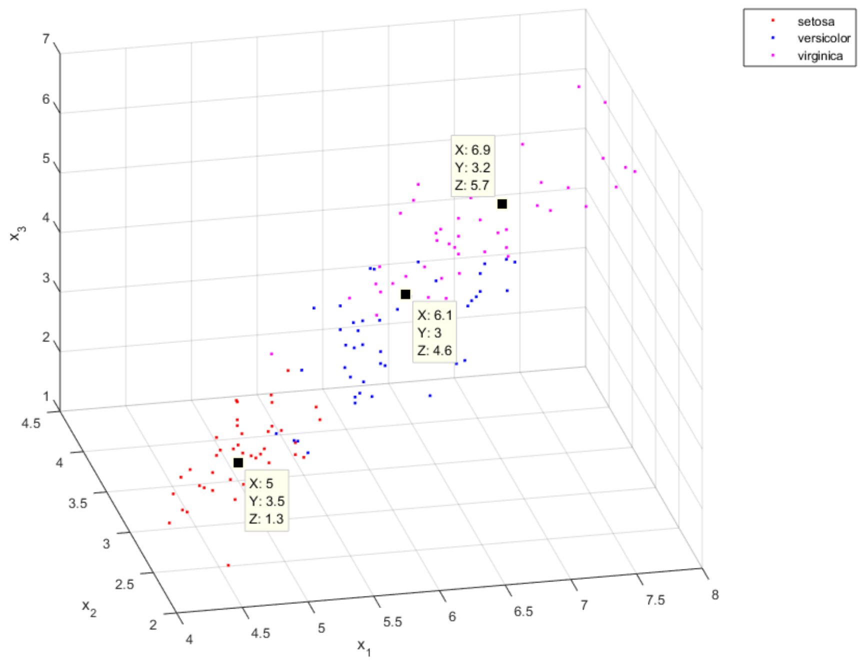
$$\boldsymbol{v}_1 = \begin{bmatrix} 5 \\ 3.5 \\ 1.3 \end{bmatrix} \quad \boldsymbol{v}_2 = \begin{bmatrix} 6.1 \\ 3 \\ 4.6 \end{bmatrix} \quad \boldsymbol{v}_3 = \begin{bmatrix} 6.9 \\ 3.2 \\ 5.7 \end{bmatrix}$$

Using the first vector as a reference, we can compute a basis:

$$\boldsymbol{u}_1 = \boldsymbol{v}_2 - \boldsymbol{v}_1 = \begin{bmatrix} 1.1 \\ -0.5 \\ 3.3 \end{bmatrix} \qquad \boldsymbol{u}_2 = \boldsymbol{v}_3 - \boldsymbol{v}_1 = \begin{bmatrix} 1.9 \\ -0.2 \\ 4.4 \end{bmatrix}$$

We can therefore conclude that the points are roughly aligned with the subspace $\mathcal{S}$, where

$$\mathcal{S} = \operatorname{span}\left(\begin{bmatrix} 1.1 \\ -0.5 \\ 3.3 \end{bmatrix}, \begin{bmatrix} 1.9 \\ -0.2 \\ 4.4 \end{bmatrix} \right)$$



The intuition here is that if we projected our data onto this subspace, we wouldn't lose much information. Although it is of little consequence for this example, you might imagine a scenario where the data is in $\mathbb{R}^{10^6}$ and there is a great computational benefit to finding a low-dimensional representation of the data.

f) Use this subspace to find a 2-dimensional classification rule. What is the average classification error in this case?

SOLUTION: We will use least-squares again, but this time with the projected data. To project our data onto the two-dimensional subspace, let's begin by finding an orthonormal basis. We can use our Gram-Schmidt code for this, and apply it to the subspace found in part (e). The result is:

$$\boldsymbol{U} = \begin{bmatrix} 0.3130 & 0.6503 \\ -0.1423 & 0.7527 \\ 0.9390 & -0.1027 \end{bmatrix}$$

Therefore, if we are given a feature vector $\boldsymbol{v} \in \mathbb{R}^3$, its coordinates in the projected space are $\tilde{\boldsymbol{v}} = \boldsymbol{U}^T\boldsymbol{v} \in \mathbb{R}^2$. Since the feature vectors are stored as rows of $\boldsymbol{A}$, the relevant transformation is $\tilde{\boldsymbol{v}}^T = \boldsymbol{v}^T\boldsymbol{U}$. So our new $\boldsymbol{A}$ matrix (with projected features) is therefore $\boldsymbol{AU} \in \mathbb{R}^{150\times 2}$. To train a classifier on the projected features, we must solve the least-squares problem:

$$\text{minimize} \quad \|\boldsymbol{AUw} - \boldsymbol{b}\|_2^2$$

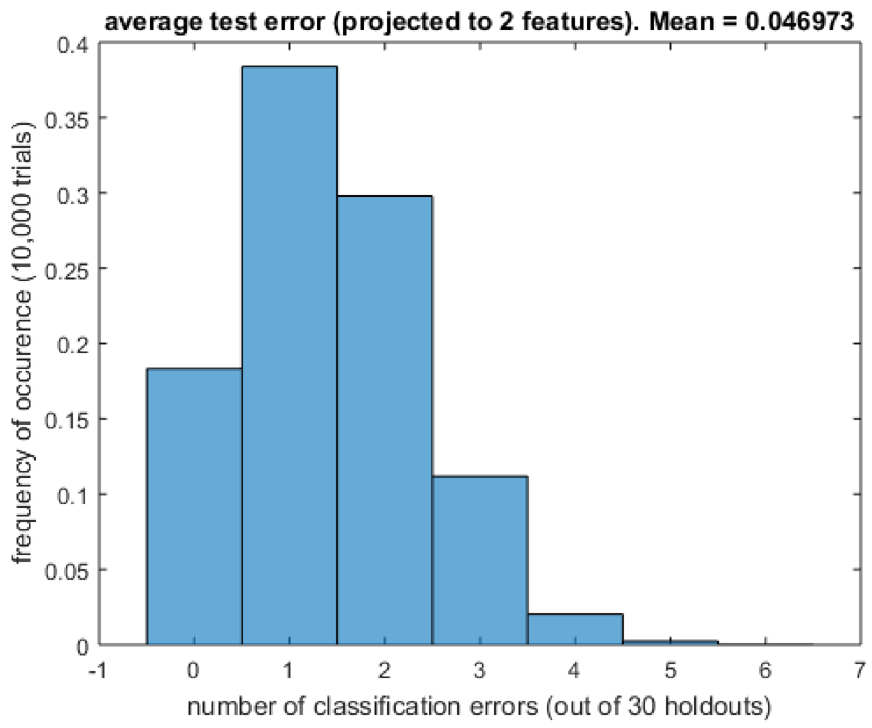
Once we have solved this to find the weight vector $\widehat{\boldsymbol{w}}$, the classification procedure is:

(a) Take the vector of 3 features $\boldsymbol{v}^T$ and compute its projection $\tilde{\boldsymbol{v}}^T = \boldsymbol{v}^T\boldsymbol{U}$.

(b) Multiply the projected feature vector by the optimal weights: $s = \tilde{\boldsymbol{v}}^T\widehat{\boldsymbol{w}}$

(c) Threshold. Depending on whether s is closer to -1, 0, or 1, assign it the label *setosa*, *versicolor*, or *virginica*.

This procedure is actually equivalent to just replacing $\boldsymbol{A}$ with $\boldsymbol{AU}$ and proceeding as in part (d). I re-ran the code from part (d) with this modification and I obtained the following error distribution:



In contrast with part (d), the error has *dropped*! We are now at 4.7% instead of the 5.9% we obtained in part (e) when we used three features.

It is worth pointing out that this sort of reduction in error is not typical. It's actually easy to prove that for any orthogonal matrix $\boldsymbol{U}$, we have $\min_{\boldsymbol{x}} \|\boldsymbol{Ax} - \boldsymbol{b}\|_2 \leq \min_{\boldsymbol{w}} \|\boldsymbol{AUw} - \boldsymbol{b}\|_2$. Therefore, projecting your data onto a lower dimensional subspace before solving the least-squares problem can ***never*** improve the residual. So what is going on here? Our classifier is not actually measuring the residual! We are measuring the *error rate*, so there is an additional thresholding operation that occurs (when we round the values to -1, 0, or 1, depending on which is closer). As it turns out, it's possible that projecting the data can cause the error rate to decrease, even though the residual will increase.