Homework 1

By: Daniel Speiser and Gideon Glass

September 15, 2015

Contents

1 Part 1: Digit Recognition

1.1 Summary

While we hypothesized an approximate 90% accuracy rate, our solution to Kaggle's digit recognizer problem (as naïve of an approach as it may be), to our surprise, is upwards of 96% accurate. Regardless of its efficiency and simplicity, the k nearest neighbor algorithm is (at least for this case) fairly precise and stable. The goal was to train the machine using the training set in order to predict all digits within the given testing set; kNN allowed us to do just that. We discovered how long these computations might take - potentially hours given the size of the data sets - and grew an appreciation for the effort and efficiency changes it takes in order for companies to deliver near or real time immediate results.

1.2 Problem Definition

Given two sets of data, train the machine to recognize approximately what pixel measurements of each digit should look like, and predict the labels of unknown digits based on these measurements.

1.3 Solution Steps

A. The data set was parsed using the pandas Python library.

B. Next, we must find one instance of each digit. To do this, we store a list of size 10 (one for each digit) of boolean flags to determine whether a certain digit has been found. Upon each digit's discovery, store the index of its first occurrence, and set the corresponding digit flag to True, indicating that it has been found. Once all 10 digits have been found, we reshape the 784 1-D array into a 28x28 2-D array, and display them, as seen below in Fig. 1. The portion of code that produces these images can be seen in Fig. 2.

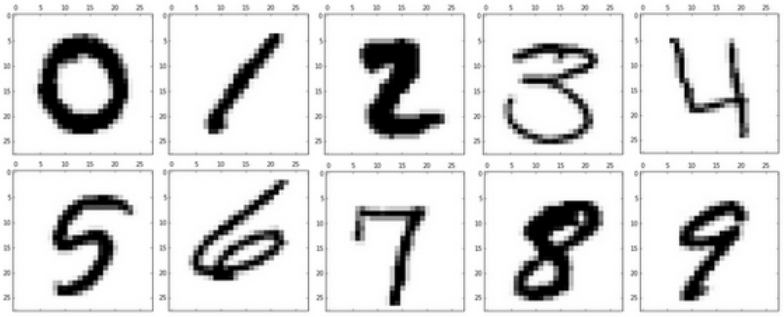

Figure 1: First of Each Digit

```
def get_digit_info (labels):
    digit_flag, digit_freq, first_instance_idx = np.zeros(10),
        np.zeros(10), {}
    for i, label in enumerate(labels):
        digit_freq[labels[i]] += 1
        if digit_flag[label] == False:
            digit_flag[label] = True
            first_instance_idx[label] = i
    return (first_instance_idx, dict(zip(range(10), digit_freq /
        sum(digit_freq))))
# Display MNIST digits
def display_digit (digit, label):
    d = digit.reshape(28,28)
    plt.matshow(d, cmap='gray_r')
    img.imsave('digit_' + str(label), d, cmap="gray_r")
    return
```

Figure 2: Find and Display Sample Digits

C. The prior probability of the classes (based on the digit frequency) is not uniform. The histogram below (figure 3) depicts the uneven distribution among the classes. Several digits, namely 1, 3, and 7 appear a few hundred times more than the other digits. The code which computed this histogram can be found in Fig. 4.

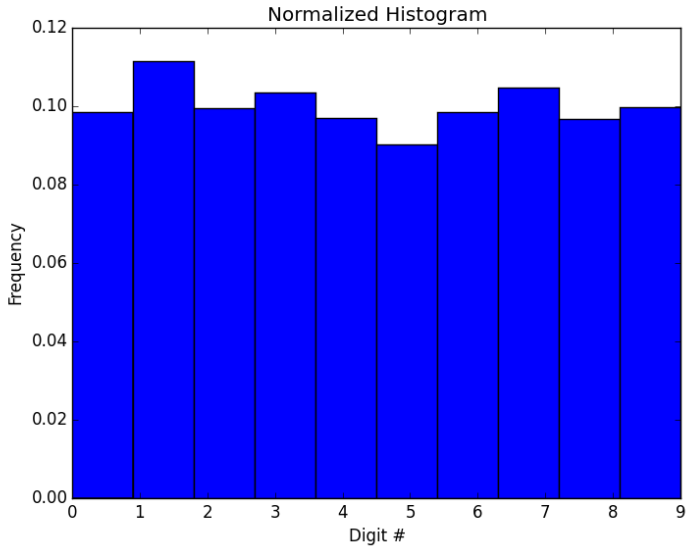

Figure 3: Digit Histogram

```
def build_normalized_histogram(digit_freq, labels):
    fig = plt.figure()
    plt.hist(digit_freq.keys(), weights=digit_freq.values())
    plt.title('Normalized Histogram')
    plt.xlabel('Digit #')
    plt.ylabel('Frequency')
    plt.gca().set_xlim([0, 9])
    fig.savefig('norm_hist.png')
    return
```

Figure 4: 'build_normalized_histogram' Function

D. Using the digits in Fig 1 as our sample digits, we obtained their best fits by computing the L2 euclidean distance between each sample, and every other digit in the data set. The resulting pairs, given with their L2 distances, can be seen in figure 5, and the portion of code that generated these images can be seen in figure 6.

Sample Digit	Best Fit	L2 Distance (In Pixels)
		1046.59543282
		489.679487012
		1380.8772574
		1834.63620372
		1356.88098225
		1066.36766643
		1446.51132038
		863.501013317
		1593.777588
		910.576740314

Figure 5: Sample digits and their corresponding best fits

```
# Find the best matches
def find_best_matches(first_instance, digits, labels):
    best_fits = [(sys.maxint, -1)] * 10 # array containing a tuple
        (distance, idx) for each digit
    for idx, digit in enumerate(digits): # for each matrix of pixels
        if idx != first_instance[labels[idx]]: # if not first instance
            of this digit
            L2 =
                spatial.distance.euclidean(digits[first_instance[labels[idx]]],
                digit) # # Distance = euclidean(first_instance_matrix,
                pixel_matrix[idx])
            if L2 < best_fits[labels[idx]][0]: # if current distance <
                current smallest distance
                best_fits[labels[idx]] = (L2, idx) # set new distance as
                    smallest
    return best_fits

best_fits = find_best_matches(indices, train_digits, train_labels)

for digit, fit in enumerate(best_fits):
        print "L2 distance between sample {0} and nearest neighbor: {1}
            pixels".format(digit, fit[0])
```

Figure 6: 'find_best_matches' Function

E. To compute the pairwise distances, we needed to take 4 cases into consideration: '0's against '0's, '1's against '1's, '0's against '1's, and '1's against '0's. The first two sets of distances are known to be our 'genuine' distances, and the last two are the 'impostor'. Once we computed these metrics, we created histograms of both the genuine pairs and the impostor pairs as can be seen in figure 7 (code in figure 8). The region of overlap as you sweep across the distance parameter represents the False Positive Rate (FPR). Ideally, the overlapping region should be minimized to prevent false positives or negatives.

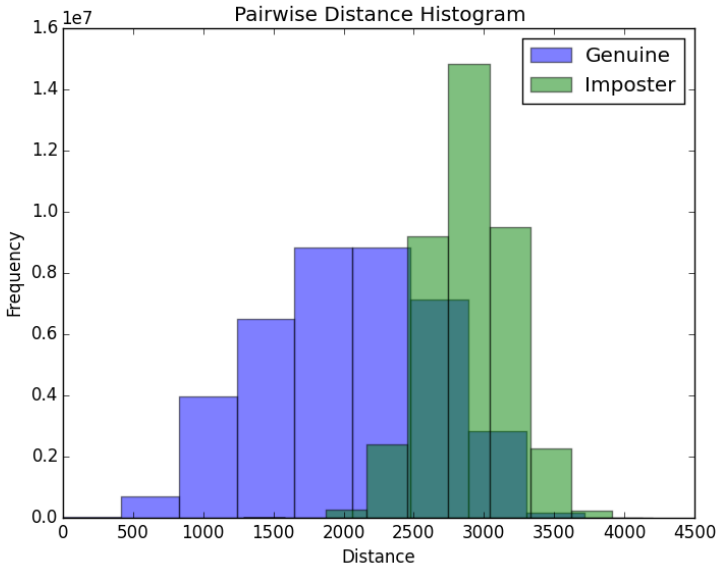

Figure 7: Pairwise Distance Histogram

```
# Get pairwise distance and plot histograms
def get_pairswise_distance (digits, labels):
    binary_classifier = {"zeros": [], "ones": []}

    for idx, digit in enumerate(digits):
        # Only concerned with digits 0 and 1
        if labels[idx] > 1:
            continue
        elif labels[idx] == 0:
            binary_classifier['zeros'].append(digit)
        else:
            binary_classifier['ones'].append(digit)

    # Genuine values
    genuine_zero =
        metrics.pairwise.pairwise_distances(binary_classifier['zeros']).flatten()
    genuine_one =
        metrics.pairwise.pairwise_distances(binary_classifier['ones']).flatten()
    genuine_total = np.concatenate((genuine_zero, genuine_one))

    # Imposter values
    imposter_zero =
        metrics.pairwise.pairwise_distances(binary_classifier['zeros'],
        binary_classifier['ones']).flatten()
    imposter_one =
        metrics.pairwise.pairwise_distances(binary_classifier['ones'],
        binary_classifier['zeros']).flatten()
    imposter_total = np.concatenate((imposter_zero, imposter_one))

    return (genuine_total, imposter_total)

def plot_pairwise_distance (genuine, imposter):
    fig = plt.figure()
    plt.hist(genuine, alpha=0.5, label='Genuine')
    plt.hist(imposter, alpha=0.5, label='Imposter')
    plt.title('Pairwise Distance Histogram')
    plt.xlabel('Distance')
    plt.ylabel('Frequency')
    plt.legend(loc='upper right')
    fig.savefig('pw_distance.png')
    return

genuine, imposter = get_pairswise_distance(train_digits, train_labels)
plot_pairwise_distance(genuine, imposter)
```

Figure 8: Pairwise Distance Functions

F. By integrating the scores (distances) of genuine and impostor pair matches, we produced the Receiver Operating Characteristic (ROC) curve. This ROC curve (figure 9) graphically represents the confusion matrix by plotting TPR against FPR. The pairwise distances between genuine and impostor points were used as "scores" to plot the ROC curve. Note, a lower

distance represents a higher score. The Equal Error Rate (EER) is the point at which there is an equal opportunity to miss classify a positive or negative sample (FPR + TPR = 1). For our data, the EER is represented by the point of intersection of the ROC curve with the green dashed line. Specifically, that point is approximately, (0.18, 0.81) (as seen in figure 10). The error rate of a classifier that guesses randomly should be approximately 50 percent, as denoted by the red dashed line in Figure 9.

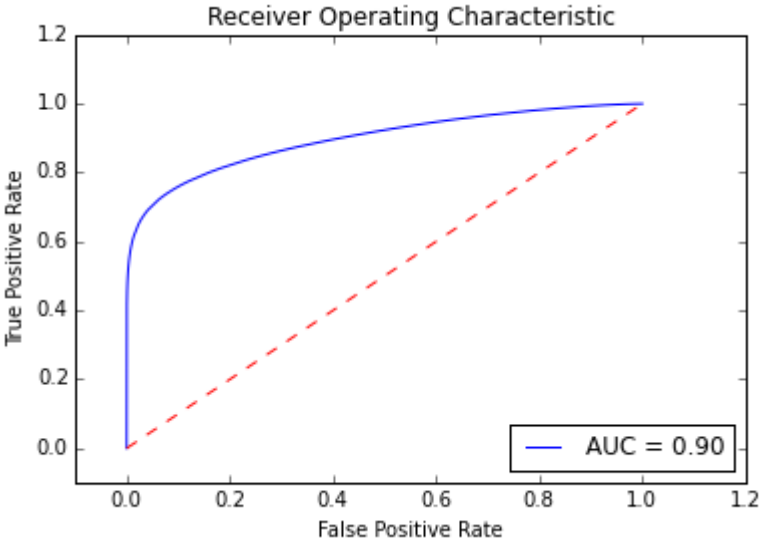

Figure 9: ROC Curve

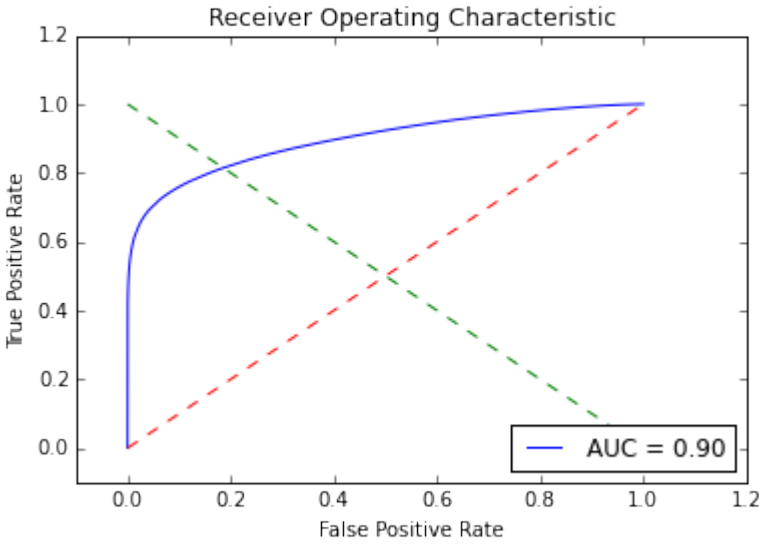

Figure 10: Equal Error Rate

```
# Plot ROC Curve
def plot_roc_curve (genuine, imposter):
    zeros = np.zeros(len(genuine))
    ones = np.ones(len(imposter))
    y_true = np.concatenate((zeros, ones))
    y_score = np.concatenate((genuine, imposter))

    # Remember - lower distance is a higher score!
    # To account for that, subtract all distances from max distance and
        plot
    m = max(y_score)
    for i in range(len(y_score)):
        y_score[i] = np.float(m - y_score[i])

    fpr, tpr, thresholds = metrics.roc_curve(y_true, y_score,
        pos_label=0)
    roc_auc = sk.metrics.auc(fpr, tpr)

    fig = plt.figure()
    plt.title('Receiver Operating Characteristic')
    plt.plot(fpr, tpr, 'b', label='AUC = %0.2f'% roc_auc)
    plt.legend(loc='lower right')
    plt.plot([0,1],[0,1],'r--')
    plt.xlim([-0.1,1.2])
    plt.ylim([-0.1,1.2])
    plt.ylabel('True Positive Rate')
    plt.xlabel('False Positive Rate')
    fig.savefig('roc_curve.png')
    return
```

Figure 11: ROC Curve Functions

G. We took the naive approach to kNN. To make an instance of our kNN classifier class, we need to pass 4 parameters: k neighbors, training data, labels for the training data, and test data. The class (found in figure 12 contains two 'get_neighbors' functions, one to obtain neighbors of the training set against itself for classification (in order to train the machine), and another which is called while predicting the digits of the test data. In our case, the prediction accuracy (as found using cross validation in the next question) was the highest with a k neighbor value of 3; after testing k = 1, 3, 5, 7, 9, 11, 13, 15, the accuracy improved from 1 to 3, peaked at 3, and slowly decreased as k increased.

```
# Implement KNN Classifier
class KNNeighborsClassifier:
    def __init__(self, k_neighbors, train_data, labels, test_data):
        self.k_neighbors = k_neighbors
        self.train_data = train_data
        self.labels = labels
        self.test_data = test_data
        return

    def get_neighbors(self, X):
        neighbors = []
        for idx1, point1 in enumerate(X): # for each matrix of pixels
            distances = []
            [distances.append((spatial.distance.euclidean(point1,
                point2), idx2)) for idx2, point2 in enumerate(X)]
             # sort list of tuples based on key 0, or distance!
            distances.sort(key=itemgetter(0))
             # remove 0 distance while comparing the same value
            nearest_neighbors = distances[1:self.k_neighbors+1]
            neighbors.append(nearest_neighbors)
        return neighbors

    def get_neighbors(self, k, train, test_inst):
        neighbors = []
        for idx, point in enumerate(train):
            neighbors.append((spatial.distance.euclidean(point,
                test_inst), idx))
        neighbors.sort(key=itemgetter(0))
        return neighbors[:k]

    def classifier(self, nearest_neighbors, digit_labels):
        possible_classes = []
        for neighbor in nearest_neighbors:
            possible_classes.append(digit_labels[neighbor[1]])

        return max(set(possible_classes), key=possible_classes.count)

    def predict(self, test_data):
        predicted_digits = []
        for test_instance in test_data:
            neighbors = self.get_neighbors(self.k_neighbors,
                self.train_data, test_instance)
            predicted_digits.append(self.classifier(neighbors,
                self.labels))
        return predicted_digits
```

Figure 12: kNN Classifier Class

H. Using a 3 fold cross-validation against our kNN classifier, we got an average of 96.661667% accuracy.

```
# Cross validation
def cross_validate (folds, digits, labels):
    cv = cross_validation.KFold(len(digits), n_folds=folds)
    results = []
    for train_idx, test_idx in cv:
        # Divide into buckets
        x_train = digits[train_idx]
        y_train = labels[train_idx]
        x_test = digits[test_idx]
        y_test = labels[test_idx]

        # Fit and predict
        kNN = KNNeighborsClassifier(3, x_train, y_train, x_test)
        prediction = kNN.predict(x_test)
        accuracy = (prediction == y_test).sum() / float(len(y_test))
        confusion_matrix = metrics.confusion_matrix(y_test, prediction)
        results.append((prediction, accuracy, confusion_matrix))
    return results
```

Figure 13: Cross Validation

I. As seen in Fig. 14, our confusion matrix is highly concentrated along the diagonal, explaining our high accuracy rating. Our classifier did struggle however when it came to a couple of pairs of digits, such as '4's and '9's, '1's and '7's, '7's and '9's, as well as '3's and '8's. Code that generated the confusion matrix and accuracy percentage can be found in fig. 15.

```
Confusion Matrix:
[4110    0    4    0    0    5   10    0    2    1]
[   0 4655    8    1    2    1    4    7    3    3]
[  31   46 3982   12    3    3    4   73   15    8]
[   6   12   30 4182    0   37    2   21   37   24]
[   3   43    0    0 3887    0   15    4    3  117]
[  14    2    1   65    2 3614   43    2   14   38]
[  26    6    0    0    5   17 4080    0    3    0]
[   2   54    9    1    8    0    0 4268    0   59]
[  20   45   10   42   14   68   17   10 3792   45]
[  14    9    2   26   38   10    2   58   20 4009]
```

Figure 14: Confusion Matrix

```
results = cross_validate(3, train_digits, train_labels)

# Print results of cross validation
predictions, accuracy, confusion_matrix = [], [], []
s = 0
for p, a, c in results:
    s += len(p)
    predictions.append(p)
    accuracy.append(a)
    confusion_matrix.append(c)

print "Number of digits: ", s
print "Number of folds: ", "3"
print "Mean accuracy: ", np.mean(accuracy)
print "Confusion Matrix: \n", sum(confusion_matrix)

cf_matrix = pd.DataFrame(sum(confusion_matrix))
file_name = "confusion_matrix_" + str(s) + '.csv'
cf_matrix.to_csv(file_name)
```

Figure 15: Cross Validation Accuracy and Confusion Matrix

J. Submitted results to kaggle, and received a score of .96929.

2 Part 2: Titanic

2.1 Summary

Unlike the Digit Recognizer data set, the TItanic data contained categorical classes and lots of missing data. The goal was to perform logistic regression and predict which passengers would have survived the disaster based on the given features. Using built in scikit-learn library methods and careful selection of input features, we performed logistic regression on training and test data and submitted our predictions to Kaggle.

2.2 Solution Steps

A. The data set was parsed using the pandas Python library.

B. The data set contained several possible features to include in the logistic regression. We decided on a combination of:

- Pclass (Passenger class)
- Age
- Sex
- Sibsp (Number of Siblings/Spouses Aboard)
- Parch Number of Parents/Children Aboard

Our hypothesis was that a correlation may exist between survival rate and these particular features. For example, we hypothesized that women and children were more likely to survive because they might be saved by others. Therefore, their age and gender play important roles in their potential survival. Additionally, family size and/or parents on board may contribute to the survival rate. For example, parents might have helped younger children survive by assisting them over themselves.

C. Some pre-processing was required to fit the data. First, the "Sex" class needed to be turned into a binary class 0,1 representing male and female respectively.

```
# Replace the categorical classes with binary
train = train_data.replace({'Sex': {'male': 0, 'female': 1}})
test = test_data.replace({'Sex': {'male': 0, 'female': 1}})
```

Figure 16: Replacing categorical classes

D. We also had to handle missing data. The age of several passengers was missing. The naive approach to "fill in the gaps" would be to replace with the mean age of the passengers. However, since we were specifically looking for a trend between males and females, we assigned the mean male age and the mean female age to the corresponding gender of the passengers that were missing the age information.

```
def replace_na (data_set):
    # Find mean male age
    male_age = data_set.loc[data_set['Sex'] == 0]['Age'].dropna()
    male_mean_age = np.mean(male_age)
    print "Mean male age: ", male_mean_age

    # Find mean female age
    female_age = data_set.loc[data_set['Sex'] == 1]['Age'].dropna()
    female_mean_age = np.mean(female_age)
    print "Mean female age: ", female_mean_age

    # Replace empty cells with appropriate mean age based on sex
    data_set[(data_set['Sex']==0) & (pd.isnull(data_set['Age']))] =
        data_set[(data_set['Sex']==0) &
        (pd.isnull(data_set['Age']))].fillna(male_mean_age)
    data_set[(data_set['Sex']==1) & (pd.isnull(data_set['Age']))] =
        data_set[(data_set['Sex']==1) &
        (pd.isnull(data_set['Age']))].fillna(female_mean_age)
    return
```

Figure 17: Dealing with missing data

E. Next, we performed cross validation (3-fold) on the training data. The mean accuracy of our predictions was approximately 79 percent.

```
# Fit
lr = linear_model.LogisticRegression(random_state=1).fit(x_train,
    y_train)
# Cross validation, k=3
scores = sk.cross_validation.cross_val_score(lr, x_train, y_train, cv=3)
print scores.mean()
0.787878787879
```

Figure 18: Dealing with missing data

F. We trained our data and then tested using the testing data set. Our predictions were exported to CSV and submitted to Kaggle, yielding a result of .74641.

```
# Predict and submit!
predicted_values = lr.predict(x_test)
p_id = test_data[['PassengerId']].values.flatten()

submission = pd.DataFrame({'PassengerId': p_id, 'Survived':
    predicted_values})
submission.to_csv('titanic_submission.csv', index=False)
print submission
```

Figure 19: Dealing with missing data

3 Part 3: Hand Written Problems

3.1 Variance of a Sum Proof

Prove that $\text{var}[X+Y] = \text{var}[X] + \text{var}[Y] + 2\,\text{cov}[X,Y]$:

One may assume that the variance of a sum is similar to the sum of an additive function, such as $\text{f}(x+y) = \text{f}(x) + \text{f}(y)$, however that is not the case. It is given that the variance of X, namely $\text{var}[X]$ is equal to $\text{E}[(X-\mu)^2]$, or the expectation of X minus the mean, squared. This is equivalent to $\text{var}[X] = \text{E}[(X - \text{E}[X])^2]$.

Now let's apply that to the variance of a sum:

$$\text{var}[X+Y] = \text{E}[((X+Y) - \text{E}[X+Y])^2]$$

This can be simplified further, since $\text{E}[X+Y]$ is a linear map!

$$\text{E}[X+Y] = \text{E}[X] + \text{E}[Y]$$

Therefore, $\text{var}[X+Y] = \text{E}[((X+Y) - (\text{E}[X] + \text{E}[Y]))^2]$. Next we can combine the X parameters into one group, and likewise with the Y's. Once they are in

pairs, we can then foil:

$$\begin{aligned}\text{var}[X+Y] &= \text{E}[((X-\text{E}[X])+(Y-\text{E}[Y]))^2] \\ &= \text{E}[((X-\text{E}[X])+(Y-\text{E}[Y]))((X-\text{E}[X])+(Y-\text{E}[Y]))] \\ &= \text{E}[(X-\text{E}[X])^2+(Y-\text{E}[Y])^2+2((X-\text{E}[X])(Y-\text{E}[Y]))] \\ &= \text{E}[(X-\text{E}[X])^2]+\text{E}[(Y-\text{E}[Y])^2]+2\,\text{E}[(X-\text{E}[X])(Y-\text{E}[Y])]\end{aligned}$$

This can now be simplified to the form we're looking for, since:

$$\begin{aligned}&\text{var}[X] = \text{E}[(X-\text{E}[X])^2], \\ &\text{var}[Y] = \text{E}[(Y-\text{E}[Y])^2], \text{ and} \\ &\text{cov}[X,Y] = 2\,\text{E}[(X-\text{E}[X])(Y-\text{E}[Y])]\end{aligned}$$

Hence, $\boxed{\text{var}[X+Y] = \text{var}[X]+\text{var}[Y]+2\,\text{cov}[X,Y]}$.

3.2 Bayes' Rule for Medical Diagnosis

Given Bayes' Theorem: $\text{P}(\theta|X) = \dfrac{\text{P}(\theta)\,\text{P}(X|\theta)}{\text{P}(X)}$

Let:

θ = Disease and $\text{P}(\theta)$ = Likelihood of having the disease = $\dfrac{1}{1000} = .0001$

X = Test Results and $\text{P}(X|\theta)$ = Probability of Accurate Test = .99

$\text{P}(X) = \text{P}(\theta)\,\text{P}(X|\theta) + \overline{\text{P}(\theta)}\overline{\text{P}(X|\theta)} = .0001 * .99 + .9999 * .01 = .010098$

Therefore, using Bayes' rule:

$$\text{P}(\theta|X) = \frac{\text{P}(\theta)\,\text{P}(X|\theta)}{\text{P}(X)} = \frac{.0001 * 99}{.010098} = \boxed{.009803921569}$$

Surprisingly, there's less than a 1% chance that the patient has the disease, even if the test is 99% accurate!

3.3 Gradient and Hessian of log-likelihood

3.3.1 (A) Derivative of Sigmoid Proof

Let $\sigma(a) = \dfrac{1}{1+\mathrm{e}^{-a}}$, show that $\dfrac{d\sigma}{da} = \sigma(a)(1-\sigma(a))$:

$$\begin{aligned}
\frac{d\sigma}{da} &= \frac{d}{da}\Big(\frac{1}{1+\mathrm{e}^{-a}}\Big) \\
&= \frac{d}{da}\big(1+\mathrm{e}^{-a}\big)^{-1} \\
&= -\big(1+\mathrm{e}^{-a}\big)^{-2}\big(-\mathrm{e}^{-a}\big) \\
&= \frac{\mathrm{e}^{-a}}{\big(1+\mathrm{e}^{-a}\big)^2} \\
&= \frac{\mathrm{e}^{-a}}{1+\mathrm{e}^{-a}} \bullet \frac{1}{1+\mathrm{e}^{-a}} \\
&= \frac{\big(1+\mathrm{e}^{-a}\big)-1}{1+\mathrm{e}^{-a}} \bullet \sigma(a) \\
&= 1 - \frac{1}{1+\mathrm{e}^{-a}} \bullet \sigma(a) \\
&= \big(1-\sigma(a)\big)\cdot\sigma(a) \\
&= \boxed{\sigma(a)\big(1-\sigma(a)\big)}
\end{aligned}$$

3.3.2 (B) Derive Gradient of Log-likelihood

Show that $\dfrac{\partial \ell(\beta)}{\partial \beta} = \sum_{i=1}^{N} x_i\big(y_i - \mathrm{P}(x_i;\beta_i)\big)$:

Given that $\ell(\beta) = \sum_{i=1}^{N}\big\{y_i\beta^T x_i - \log\big(1+\mathrm{e}^{\beta^T x_i}\big)\big\}$, we can derive the above equation.

$$\begin{aligned}
\ell(\beta) &= \sum_{i=1}^{N}\big\{y_i\beta^T x_i - \log\big(1+\mathrm{e}^{\beta^T x_i}\big)\big\} \\
\frac{\partial \ell(\beta)}{\partial \beta} &= \sum_{i=1}^{N}\Big\{\frac{\partial}{\partial\beta}\big(y_i\beta^T x_i\big) - \frac{\partial}{\partial\beta}\Big(\log\big(1+\mathrm{e}^{\beta^T x_i}\big)\Big)\Big\}
\end{aligned}$$

Let's look and simplify these interior partial differentials separately:

$$\frac{\partial}{\partial\beta}\big(y_i\beta^T x_i\big) = y_i\cdot 1\cdot x_i = x_i y_i$$

and

$$\begin{aligned}\frac{\partial}{\partial\beta}\Big(\log\big(1+\mathrm{e}^{\beta^T x_i}\big)\Big) &= \frac{1}{1+\mathrm{e}^{\beta^T x_i}}\cdot\mathrm{e}^{\beta^T x_i}\cdot x_i \\ &= \frac{\mathrm{e}^{\beta^T x_i}}{1+\mathrm{e}^{\beta^T x_i}}\cdot x_i \\ &= \mathrm{P}(x_i;\beta)\cdot x_i \\ &= x_i\,\mathrm{P}(x_i;\beta)\end{aligned}$$

Now we can plug these simplified differentials back in to get the derived formula we were looking for:

$$\begin{aligned}\frac{\partial\ell(\beta)}{\partial\beta} &= \sum_{i=1}^{N}\big\{x_i y_i - x_i\,\mathrm{P}(x_i;\beta)\big\} \\ &= \boxed{\sum_{i=1}^{N}\big\{x_i\big(y_i-\mathrm{P}(x_i;\beta)\big)\big\}}\end{aligned}$$

3.3.3 (C) Positive Definite Hessian Matrix

Prove that the Hessian matrix for the log likelihood, $\dfrac{\partial^2\ell(\beta)}{\partial\beta\partial\beta^T} = -\boldsymbol{X}^T\boldsymbol{W}\boldsymbol{X}$, is positive definite:

To prove that the result $\boldsymbol{X}^T\boldsymbol{W}\boldsymbol{X}$ is positive definite, we must prove that $\boldsymbol{X}^T\boldsymbol{W}\boldsymbol{X} > 0$ for all vectors $x \neq 0$.

We are given that:

$X \epsilon \mathbb{R}^{N\times(P+1)}$is the matrix of all the feature vectors x_i

$W \varepsilon \mathbb{R}^{N\times N}$is a diagonal weighing matrix containing the values of $\mathrm{P}(x_i;\beta)(1-\mathrm{P}(x_i;\beta))$ on the diagonal

Since $\boldsymbol{W}$ is a diagonal matrix, we know by definition that there are non-zero values along the diagonal. Therefore, we know that $\boldsymbol{W}$ has diagonal values of $0 < \mathrm{P}(x_i;\beta)(1-\mathrm{P}(x_i;\beta)) \leqslant 1$.

Now, let's look at the results of the matrix multiplication of the below matrices, named $\boldsymbol{X}^T$, $\boldsymbol{W}$, and $\boldsymbol{X}$ respectively:

$$\begin{bmatrix} x_{0,0} & \cdots & \cdots & \cdots & x_{N,0} \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ x_{0,P+1} & \cdots & \cdots & \cdots & x_{N,P+1} \end{bmatrix} \begin{bmatrix} w_{0,0} & 0 & \cdots & \cdots & 0 \\ 0 & w_{1,1} & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & 0 \\ 0 & \cdots & \cdots & 0 & w_{N,N} \end{bmatrix} \begin{bmatrix} x_{0,0} & \cdots & \cdots & \cdots & x_{P+1,0} \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ x_{0,N} & \cdots & \cdots & \cdots & x_{P+1,N} \end{bmatrix}$$

Where $w_{i,j} = \mathrm{P}(x_i;\beta)(1-\mathrm{P}(x_i;\beta))$

After $\boldsymbol{X}^T$ and $\boldsymbol{W}$ are multiplied, the resulting matrix (which we'll call $\boldsymbol{R}$) is a matrix of size $N \times (P+1)$ weighted along the diagonal with values of

$x_i \cdot \mathrm{P}(x_i;\beta)(1-\mathrm{P}(x_i;\beta))$. When we then multiply the resulting matrix $\boldsymbol{R}$ with $\boldsymbol{X}$, we get a matrix (which we'll name $\boldsymbol{R}_2$) with values $x_i^2 \cdot \mathrm{P}(x_i;\beta)(1-\mathrm{P}(x_i;\beta))$ along the diagonal. Since x_i^2 is always positive, and it was given (by definition) that $0 < \mathrm{P}(x_i;\beta)(1-\mathrm{P}(x_i;\beta)) \leqslant 1$, we know that the result $\boldsymbol{R}_2$ is > 0 for all vectors $X \neq 0$, therefore, it is positive definite.

References

[1] Matplotlib Plotting (Histograms, pixel images, etc): http://matplotlib.org/

[2] NumPy Arrays (reshape, etc.): http://www.numpy.org/

[3] Pandas Data Parser (read_csv, to_csv): http://pandas.pydata.org/

[4] Scikit-learn Cross Validation, KFolds, and Linear Regression: http://scikit-learn.org/

[5] SciPy Spatial Euclidean Distance: http://www.scipy.org/