# AMATH 582 Homework 4: Classifying Digits

Brady Griffith

**Abstract**

In this project handwritten digits from the MNIST data set are transformed into the 100 most important PCA modes. This is then put through three differemt clasification algorithms: linear discriminant analysis, support vector machines, and decision trees. The performance is then compared.

## Introduction and Overview

The MNIST database contains 60,000 handwritten digits from 250 different writers. Half come from high school students and half from census workers. This set is a popular way of comparing different machine learning techniques. In project, I will look at the linear discriminant analysis (LDA), for differentiating between digits in sets that contain either two or three. I will also look at how two more sophisticated algorithms, support vector machines (SVM) and decision tree classifiers preform in comparison.

## Theoretical Background

Before I preform any analysis, it is preferable to reduce the order of the image vectors, and switch into an orthonormal basis. This is exactly the job the SVD preforms. The last lab discussed at length how this process works, so I will skip over the details here. The results decomposes the data matrix $\mathbf{X}$ into three matrices

$$\mathbf{U\Sigma V} = \mathbf{X}$$

$\mathbf{V}$ has columns of the orthonormal basis for $\mathbf{X}$. $\mathbf{\Sigma}$ is the strength of this projection, with larger diagonals implying that the coresponding column of $\mathbf{V}$ is more important to properly representing $\mathbf{X}$. And $\mathbf{U}$ .

LDA makes a differentiation by projecting the data onto an axis which maximizes the distance between means of the two classes [1]. This axis $\mathbf{w}$ can be defined as

$$\mathbf{w} = \arg\max_{\mathbf{w}} \frac{\mathbf{w}^T \mathbf{S}_B \mathbf{w}}{\mathbf{w}^T \mathbf{S}_B \mathbf{w}}$$

where

$$\mathbf{S}_B = (\mu_2 - \mu_1)(\mu_2 - \mu_1)^T$$

and

$$\mathbf{S}_W = \sum_{j=1}^{2} \sum_{\mathbf{x}} (\mathbf{x} - \mu_j)(\mathbf{x} - \mu_j)^T$$

with $\mu_j$ being the means of the cluseter in each class. This form of problem can be solved as a generalized eigenvector problem.

$$\mathbf{S}_B\mathbf{w} = \lambda\mathbf{S}_W\mathbf{w}$$

When projecting the data vector onto $\mathbf{w}$ the vlaue taken by each class will tend to be fall around two different centers. Simply declaring a threshold in the middle will allow for classification.

This project also explores two different classification techniques. To fully explain them is beyond the score of this report, and I would direct the reader to the scikit-learn package for more information [2]. I will instead explain at a very high level.

Support Vector Machines work by dividing up the data vector space into the number of categories desired. The linear version used in this project does this by marking three centers and choosing the category whose center the data is closest to. The fitting process involves moving these centers to best match the training data.

A decision tree classifier creates a tree of conditions on the data vector. At the end of the tree of conditions each branch has one classification. The fitting process involves defining these condition along with the number needed. This method has the advantage of being easy to interpret the model created.

# Algorithm Implementation and Development

The SVD is preformed using the numpy numpy.linalg.svd function. For the rest of the project, the algorithms are applied to the data projected onto the first 100 columns of $\mathbf{V}$.

Mimicking the style built into the scikit-learn package, all of the clasification algorithms are built as objects with two functions. The Classifier . fit (X, y) function is used to train the model. The data is provided as rows in X, and the labels in the array y. I implement LDA for classifying into two or three categories. For all classification problems, I train using a set of $N$ digit samples, and the performance reported comes from a set of $N/5$ samples excluded from the training data.

In the case of two categories, I apply LDA as described in the Theoretical Background section. In the case of three, I apply LDA 3 times, to all combinations of the three labels. I then take the classification which was chosen by the most of the three. If all three disagree, I randomly choose a label. The idea behind this method is that for the two combinations where the correct answer is compared, it will be selected. In the 3rd, the result will be nonsense, but can be ignored. A disadvantage of this method is that the number of times LDA must be preformed grows as $\mathcal{O}(n^2)$, where $n$ is the number of digits. For all 10 digits, LDA would need to be preformed 45 times.

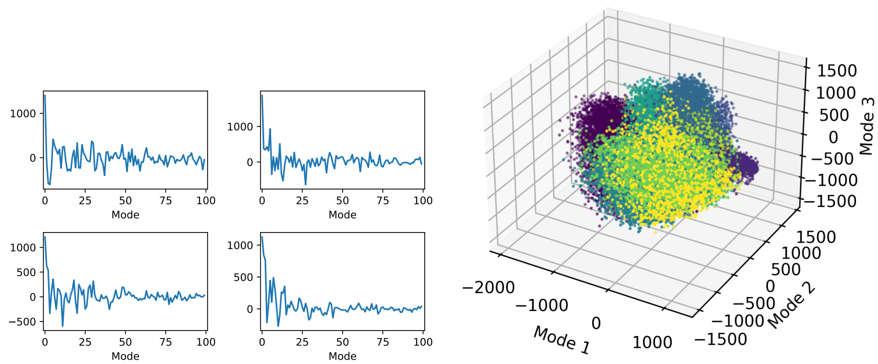


Figure 1: Left: 4 Examples of the spectrum of the digit samples in the SVD modes. Right) All of the digit samples projected onto the 2nd, 3rd and 4th SVD modes. Each digit is colored differently.

SVM and tree classifier are prefomed using the objects build into the scikit-learn package. It is worth noting how east to implement these were. This analysis can be added into future projects with minimal effort.

# Computational Results

The digits are projected onto the SVD modes. Some examples of the spectrum in these modes is given in fig 1. All the digits examples are projected onto the 2nd, 3rd and 4th modes. The digits visually start to cluster, which is an important requirement from the clasification algorithms that will follow.

It is not necessary to use the full set of modes. Figure 2 plots the total fraction of mode power remaining after $N$ modes are kept. By 100 modes, 98% of the power has already been collected. If you truncate there, the numbers are still easily readable.

Once in the reduced order modes, LDA is applied identify between pairs of digits. Each pair is trained using 2000 samples of the digits. The error rate is reported in firgure 3. 4 and 7 were the easiest to differentiate and 3 and 5 the hardest. I use this to inform the sets of 3 digits for the 3 classification test. The first set is composed of numbers that were all easily distinguised, 0, 2 and 8. This is the best case test. The more sifficult test uses 3 commonly confused numbers, 0, 4, 5. For the easy set the error rate was 35% and for the difficult set the error rate was 40%. This is much better than chance, but I wouldn't stake my postage delivery on it.

The same test differentiating between all combinations of digit pairs is preformed again with two more advanced alogorithms. The error rates are presented in fig

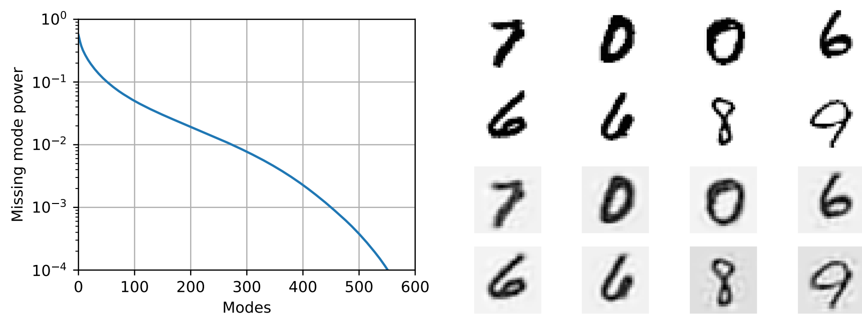


Figure 2: Left: The fraction of total power that is still not included after N modes are included. Right: On top are 8 selected digit samples, and on the bottom are the same digits, represented with 100 SVD modes.

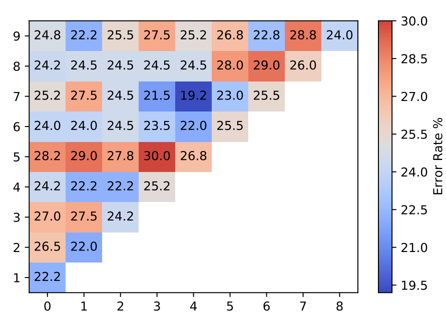


Figure 3: For all pairings of digits, the error rate of LDA differentiating the two.

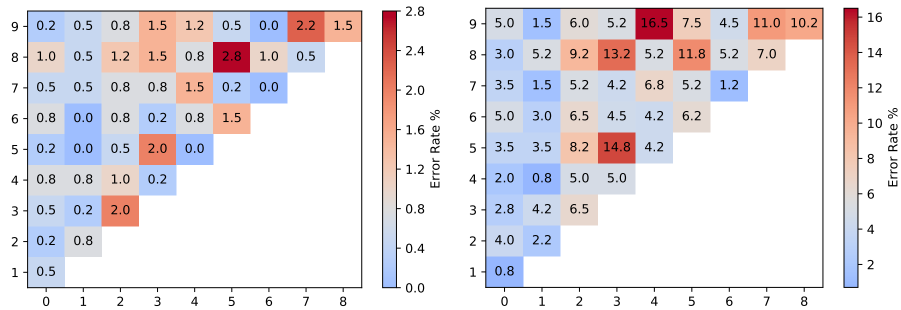

Figure 4: For all pairings of digits, the error rate differentiating the two. On the left using SVM and the right a decision tree.

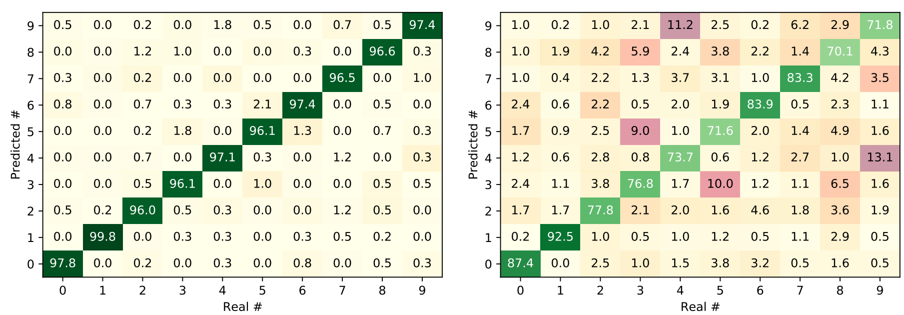

Figure 5: For the correct digit, the percetage that it was identified as. On the left using SVM and the right a decision tree.

4. Both models preform much better than LDA, but SVD is the clear winner. Both models struggle with the oairs (5, 3) and (5, 8). The decision tree struggles more with (4, 9).

When applied to all 10 digits, some of these features persist. The 9 and 4 confusion severely hurts the ability of the decision tree to correctly label both of those digits. Again SVM preforms better, scoring in the high 90s for most digits.

## Summary and Conclusions

Digit differentiation is explored with three different algorithms. SVMs are the best performer on this data set. The biggest take away from this project for me is how easy it is to implement models using scikit-learn. There is no reason that these shouldn't be tried out on data sets in the research.

## References

[1] J. N. Kutz, *Data-driven modeling & scientific computation: methods for complex systems & big data*. Oxford: Oxford University Press, first edition ed., 2013. OCLC: ocn858608087.

[2] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

Here is a link to the Github repository for this project.

# Python Functions

## LDA Objects

```
class LDA()
```

Linear Discrimination Analysis model for classifying in up to 3 groups.

#### fit

```
| fit(X, y)
```

Trains the model.

**Arguments**:

- `X` *array-like* - Contains rows of the training data examples.
- `y` *array-like* - Contains labels for the training data rows.

#### predict

```
| predict(X)
```

Predicts the category of rows of X.

**Arguments**:

- `X` *array-like* - Contains rows of the data to categorize.

## evaluation

### NaiveClassifier Objects

```
class NaiveClassifier()
```

A model classifier that randomly guesses a digit.

This was created as a simple test article to make sure the number_confusion code worked indpendent of any model used.

#### number_confusion

```
number_confusion(model, train_n, V)
```

Plots a how the model preforms at distinguising pairs of digits.

**Arguments**:

- `model` - The model class. Should have functions fit(X, y) that trains the model to identify labels y using data X and predict(X) that will label data in the matrix X.

- `train_n` *int* - The number of example digits to train on.
- `V` *array-like* - A matrix to transform the data into the basis for predictions.

#### full_classification

`full_classification(model, train_n, V)`

Plots a how the model preforms at identifying digits.

**Arguments**:

- `model` - The model class. Should have functions fit(X, y) that trains the model to identify labels y using data X and predict(X) that will label data in the matrix X.
- `train_n` *int* - The number of example digits to train on.
- `V` *array-like* - A matrix to transform the data into the basis for predictions.

#### digit_performance

`digit_performance(model, train_n, V, digits)`

Plots a how the model preforms at identifying digits.

**Arguments**:

- `model` - The model class. Should have functions fit(X, y) that trains the model to identify labels y using data X and predict(X) that will label data in the matrix X.
- `train_n` *int* - The number of example digits to train on.
- `V` *array-like* - A matrix to transform the data into the basis for predictions.
- `digits` *list* - List of digits to test on

### main

#### run_analysis

`run_analysis()`

Runs the full analysis for the MNIST handwritting project

### svd

#### plot_mode_proj

`plot_mode_proj(X, V, labels, modes)`

Creates a 3D projection of X into the 3 selected SVD modes

**Arguments**:

- `X` *array_like* - Data matrix with rows of images
- `V` *array_like* - Matrix with mode vectors as columns

- `modes` *list* - List of 3 mode indexes to project on

#### plot_n_modes

```
plot_n_modes(X, V, n)
```

Shows the numbers represented with the selected number of SVD modes

**Arguments**:

- `X` *array_like* - Data matrix with rows of images
- `V` *array_like* - Matrix with mode vectors as columns
- `n` *int* - Number of modes to use in the representation

#### plot_svd_spectrum

```
plot_svd_spectrum(X, V)
```

Plots the svd spectrum of 4 random images.

**Arguments**:

- `X` *array_like* - Data matrix with rows of images
- `V` *array_like* - Matrix with mode vectors as columns

#### plot_mode_fraction

```
plot_mode_fraction(s)
```

Plots the fraction of power represented with n modes

**Arguments**:

- `s` *array-like* - 1D arrray of the variances of the principal components.

### loadmnist

#### load_data

```
load_data(numbers=None, size=None)
```

Loads a matrix of selected numbers.

Creates a matrix of shape (nsamples, npixels) where nsamples is the number of occurrences of the selected numbers.

**Arguments**:

- `numbers` *list* - A list of digits to load. If None or empty, all digits will by loaded. Defaults to None.
- `size` *int* - The max number of images to load. If None, all images will be loaded. Defaults to None.

**Returns**:

- `np.int32` - Matrix with rows of images
- `np.int8` - Array of digit labels for rows of images

# Python Code

## main.py

```python
import numpy as np
from sklearn import svm, tree
import loadmnist
import svd
import evaluation
import lda


def run_analysis():
    """Runs the full analysis for the MNIST handwritting project"""
    X_large, labels_l = loadmnist.load_data()
    X_small, labels_s = loadmnist.load_data(size=10000)
    U, s, V = np.linalg.svd(X_small)
    V = V[:100]

    # svd.plot_mode_proj(X_large, V, labels_l, [1, 2, 3])
    # svd.plot_n_modes(X_large, V, 100)
    # svd.plot_svd_spectrum(X_large, V)
    # svd.plot_mode_fraction(s)

    # N = 2000
    # evaluation.number_confusion(evaluation.NaiveClassifier(), N, V)
    # evaluation.number_confusion(lda.LDA(), N, V)
    # evaluation.number_confusion(svm.SVC(), N, V)
    # evaluation.number_confusion(tree.DecisionTreeClassifier(), N, V)

    # N = 3000
    # print('Good:')
    # print(evaluation.digit_performance(lda.LDA(), N, V, [0, 2, 8]))
    # print('Bad:')
    # print(evaluation.digit_performance(lda.LDA(), N, V, [0, 4, 5]))

    N = 20000
    evaluation.full_classification(svm.SVC(), N, V)
    evaluation.full_classification(tree.DecisionTreeClassifier(), N, V)


if __name__ == '__main__':
```

```python
    run_analysis()
```

### loadmnist.py

```python
import numpy as np
from mnist import MNIST

_mnist_path = '/home/brady/Documents/class/2021w/AMATH582/HW4/data'
images_raw = None
labels_raw = None


def load_data(numbers=None, size=None):
    """Loads a matrix of selected numbers.

    Creates a matrix of shape (nsamples, npixels) where nsamples is the number
    of occurrences of the selected numbers.

    Args:
        numbers (list): A list of digits to load. If None or empty, all digits
            will by loaded. Defaults to None.
        size (int): The max number of images to load. If None, all images will
            be loaded. Defaults to None.

    Returns:
        np.int32: Matrix with rows of images
        np.int8: Array of digit labels for rows of images
    """

    global images_raw, labels_raw
    if images_raw is None or labels_raw is None:
        mndata = MNIST(_mnist_path)
        images_raw, labels_raw = mndata.load_training()

        images_raw = np.float64(images_raw)
        labels_raw = np.int8(labels_raw)

    images, labels = images_raw.copy(), labels_raw.copy()

    if numbers:
        # Select numbers if numbers isn't None or empty
        mask = np.isin(labels, numbers)
        images = images[mask]
        labels = labels[mask]

    if size:
```

```
        if len(labels) > size:
            mask = np.random.choice(len(labels), size=size, replace=False)
            images = images[mask]
            labels = labels[mask]

    return images, labels
```

### svd.py

```
import numpy as np
import matplotlib.pyplot as plt


def plot_mode_proj(X, V, labels, modes):
    """Creates a 3D projection of X into the 3 selected SVD modes

    Args:
        X (array_like): Data matrix with rows of images
        V (array_like): Matrix with mode vectors as columns
        modes (list): List of 3 mode indexes to project on
    """
    fig = plt.figure(figsize=(4, 4))
    ax = fig.add_subplot(111, projection='3d')

    Y = np.dot(V[modes], X.T)
    ax.set_xlabel(f'Mode {modes[0]}')
    ax.set_ylabel(f'Mode {modes[1]}')
    ax.set_zlabel(f'Mode {modes[2]}')
    ax.scatter(Y[0], Y[1], Y[2], c=labels, s=1)

    fig.savefig('HW4/figures/svd_projection.png', bbox_inches='tight', dpi=300)


def plot_n_modes(X, V, n):
    """Shows the numbers represented with the selected number of SVD modes

    Args:
        X (array_like): Data matrix with rows of images
        V (array_like): Matrix with mode vectors as columns
        n (int): Number of modes to use in the representation
    """
    fig, axs = plt.subplots(4, 4, figsize=(4, 3))

    selected = np.random.randint(0, X.shape[0], 8)
    Y = np.dot(V, X[selected].T)
    Z = np.dot(V[:n].T, Y[:n]).T
```

```
    for ax, image in zip(axs[:2].flatten(), X[selected]):
        ax.axis('equal')
        ax.axis('off')
        ax.imshow(np.reshape(image, (28, 28)), cmap='Greys')

    for ax, image in zip(axs[2:].flatten(), Z):
        ax.axis('equal')
        ax.axis('off')
        ax.imshow(np.reshape(image, (28, 28)), cmap='Greys')

    fig.savefig('HW4/figures/reduced_dim.png', bbox_inches='tight', dpi=300)


def plot_svd_spectrum(X, V):
    """Plots the svd spectrum of 4 random images.

        Args:
        X (array_like): Data matrix with rows of images
        V (array_like): Matrix with mode vectors as columns
    """
    fig, axs = plt.subplots(2, 2, figsize=(6, 4))

    selected = np.random.randint(0, X.shape[0], 4)
    Y = np.dot(V, X[selected].T).T

    for ax, spectrum in zip(axs.flatten(), Y):
        ax.set_xlabel('Mode')
        ax.set_xlim(-1, 101)
        ax.plot(spectrum)

    fig.tight_layout()
    fig.savefig('HW4/figures/svd_spectrum.pdf', bbox_inches='tight')


def plot_mode_fraction(s):
    """Plots the fraction of power represented with n modes

    Args:
        s (array-like): 1D arrray of the variances of the principal components.
    """
    fig, ax = plt.subplots(figsize=(4, 3))
    ax.set_xlim(0, 600)
    ax.set_ylim(1e-4,1)
    ax.set_yscale('log')
    ax.grid()
```

```python
    f = 1 - np.cumsum(s**2) / np.sum(s**2)

    ax.set_xlabel('Modes')
    ax.set_ylabel('Missing mode power')

    ax.plot(f)
    fig.savefig('HW4/figures/mode_frac.pdf', bbox_inches='tight')
```

### evaluation.py

```python
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.ticker import MaxNLocator
from itertools import combinations
import loadmnist


class NaiveClassifier:
    """A model classifier that randomly guesses a digit.

    This was created as a simple test article to make sure the number_confusion
    code worked indpendent of any model used.
    """
    def fit(self, X, y):
        self.choices = np.int8(list(set(y)))

    def predict(self, X):
        index = np.random.randint(0, len(self.choices), X.shape[0])
        return self.choices[index]


def number_confusion(model, train_n, V):
    """Plots a how the model preforms at distinguising pairs of digits.

    Args:
        model: The model class. Should have functions fit(X, y) that trains the
            model to identify labels y using data X and predict(X) that will
            label data in the matrix X.
        train_n (int): The number of example digits to train on.
        V (array-like): A matrix to transform the data into the basis for
            predictions.
    """
    error_rate = np.full((10, 10), np.nan)

    fig, ax = plt.subplots(figsize=(6, 4))
```

```python
    for digits in combinations(range(10), 2):
        X, labels = loadmnist.load_data(numbers=digits,
                                        size=train_n+(train_n//5))

        Y = np.dot(V, X.T).T

        model.fit(Y[:train_n], labels[:train_n])
        model_lables = model.predict(Y[train_n:])

        errors = np.sum(model_lables != labels[train_n:])
        e = 100*errors / len(model_lables)
        error_rate[digits[1], digits[0]] = e
        ax.text(digits[0], digits[1], f'{e:0.1f}', c='k',
                va='center', ha='center')

    ax.set_xlim(-.5, 8.5)
    ax.set_ylim(.5, 9.5)
    ax.xaxis.set_major_locator(MaxNLocator(integer=True))
    ax.yaxis.set_major_locator(MaxNLocator(integer=True))

    X, Y = np.meshgrid(np.arange(11)-.5, np.arange(11)-.5)
    m = np.nanmean(error_rate)
    r = np.nanmax(np.abs(error_rate - m))

    mesh = ax.pcolormesh(X, Y, error_rate, cmap='coolwarm', vmin=m-r, vmax=m+r)
    l = np.floor(10*np.nanmin(error_rate))/10
    r = np.ceil(10*np.nanmax(error_rate))/10
    bounds = np.linspace(l, r, 512)
    cbar = fig.colorbar(mesh, boundaries=bounds, label='Error Rate %')
    cbar.set_ticks(MaxNLocator(8))
    fig.savefig('HW4/figures/{}-digits_conf.pdf'.format(type(model).__name__),
                bbox_inches='tight')


def full_classification(model, train_n, V):
    """Plots a how the model preforms at identifying digits.

    Args:
        model: The model class. Should have functions fit(X, y) that trains the
            model to identify labels y using data X and predict(X) that will
            label data in the matrix X.
        train_n (int): The number of example digits to train on.
        V (array-like): A matrix to transform the data into the basis for
            predictions.
    """
    frac_rate = np.full((10, 10), np.nan)
```

```python
    fig, ax = plt.subplots(figsize=(6, 4))

    X, labels = loadmnist.load_data(size=train_n+(train_n//5))

    Y = np.dot(V, X.T).T
    model.fit(Y[:train_n], labels[:train_n])
    model_lables = model.predict(Y[train_n:])

    ax.set_xlabel('Real #')
    ax.set_ylabel('Predicted #')

    total = np.bincount(labels[train_n:])

    for real in range(10):
        for predict in range(10):
            n = np.sum((model_lables == predict) & (labels[train_n:] == real))
            frac = 100*n / total[real]
            frac_rate[real, predict] = frac
            c = 'w' if real == predict else 'k'
            ax.text(real, predict, f'{frac:0.1f}', c=c,
                    va='center', ha='center')

    ax.set_xlim(-.5, 9.5)
    ax.set_ylim(-.5, 9.5)
    ax.xaxis.set_major_locator(MaxNLocator(integer=True))
    ax.yaxis.set_major_locator(MaxNLocator(integer=True))

    X, Y = np.meshgrid(np.arange(11)-.5, np.arange(11)-.5)

    off_diag = frac_rate.copy()
    np.fill_diagonal(off_diag, np.nan)
    ax.pcolormesh(X, Y, off_diag, cmap='YlOrRd', alpha=.4, vmin=0, vmax=11.2)

    on_diag = np.full_like(off_diag, np.nan)
    np.fill_diagonal(on_diag, 1)
    on_diag *= frac_rate
    m = np.nanmin(on_diag) - (np.nanmax(on_diag)-np.nanmin(on_diag))*.5
    ax.pcolormesh(X, Y, on_diag, cmap='Greens', vmin=50, vmax=100)

    fig.savefig('HW4/figures/{}-classification.pdf'.format(type(model).__name__),
                bbox_inches='tight')


def digit_performance(model, train_n, V, digits):
    """Plots a how the model preforms at identifying digits.
```

```
    Args:
        model: The model class. Should have functions fit(X, y) that trains the
            model to identify labels y using data X and predict(X) that will
            label data in the matrix X.
        train_n (int): The number of example digits to train on.
        V (array-like): A matrix to transform the data into the basis for
            predictions.
        digits (list): List of digits to test on
    """
    X, labels = loadmnist.load_data(numbers=digits,
                                    size=train_n+(train_n//5))

    Y = np.dot(V, X.T).T

    model.fit(Y[:train_n], labels[:train_n])
    model_lables = model.predict(Y[train_n:])

    errors = np.sum(model_lables != labels[train_n:])
    error_rate = 100*errors / len(model_lables)
    return error_rate
```

## lda.py

```
import numpy as np
import itertools
from scipy.linalg import eig


class LDA:
    """Linear Discrimination Analysis model for classifying in up to 3 groups.
    """
    def fit(self, X, y):
        """Trains the model.

        Args:
            X (array-like): Contains rows of the training data examples.
            y (array-like): Contains labels for the training data rows.
        """
        self.digits = list(set(y))

        if len(self.digits) == 2:
            X0, X1 = (X[y == d] for d in self.digits)
            m0, m1 = (np.mean(Xi, axis=0) for Xi in [X0, X1])

            S_b = np.outer(m1 - m0, m1 - m0)
```

```
            S_w = np.zeros((len(m0), len(m0)))
            for m_j in [m0, m1]:
                for i in range(X.shape[0]):
                    S_w += np.outer(X[i] - m_j, X[i] - m_j)

            W, V = eig(S_b, S_w)
            i = np.argmax(W)
            w = V[:, i]
            self.wt = (w / np.linalg.norm(w)).T

            v0 = np.dot(X0, self.wt)
            v1 = np.dot(X1, self.wt)

            if np.median(v1) < np.median(v0):
                self.wt = -self.wt
                v0 = -v0
                v1 = -v1

            x = np.linspace(0, .5, 1024)
            p0 = np.quantile(v0, x)
            p1 = np.quantile(v0, 1-x)
            mp = np.argmin(np.abs(p1 - p0))
            self.threshold = (p0[mp] + p1[mp])/2
        elif len(self.digits) == 3:
            self.pair_ldas = []
            self.pair_digits = []
            for d_select in itertools.combinations(self.digits, 2):
                self.pair_ldas.append(LDA())
                self.pair_digits.append(d_select)
                mask = np.isin(y, d_select)
                self.pair_ldas[-1].fit(X[mask], y[mask])
        else:
            print(self.digits)
            raise RuntimeError()

    def predict(self, X):
        """Predicts the category of rows of X.

        Args:
            X (array-like): Contains rows of the data to categorize.
        """
        if len(self.digits) == 2:
            r = np.dot(X, self.wt)
            return np.where(r < self.threshold, self.digits[0], self.digits[1])
        if len(self.digits) == 3:
```

```python
    votes = np.zeros((X.shape[0], 3))
    for i, (d_select, lda) in enumerate(zip(self.pair_digits, self.pair_ldas)):
        votes[:, i] = lda.predict(X)

    counts = np.zeros((X.shape[0], 3), dtype=np.int64)
    for i in range(votes.shape[0]):
        for j in range(3):
            for k, d in enumerate(self.digits):
                if votes[i, j] == d:
                    counts[i, k] += 1

    label = np.zeros(X.shape[0], np.int8)
    for i in range(X.shape[0]):
        if np.max(counts[i]) > 1:
            label[i] = self.digits[np.argmax(counts[i])]
        else:
            label[i] = self.digits[np.random.randint(3)]

    return label
else:
    raise RuntimeError()
```