David Ouyang Moench
Data Mining [625.740]
11/16/22

Image Segmentation Using SVMs For Forest Landcover Estimation

INTRODUCTION

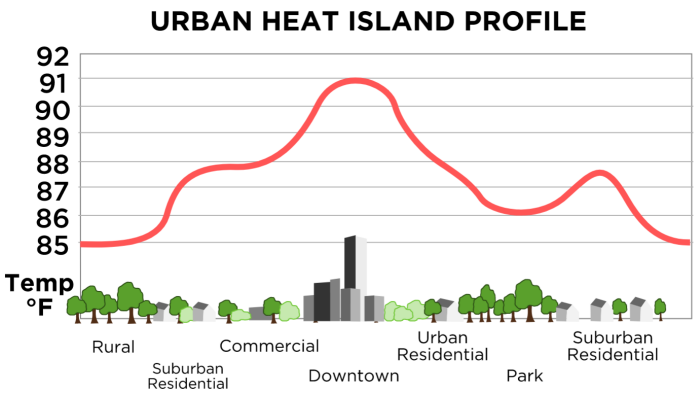
In this project I develop a binary classifier to perform image segmentation of satellite images of the Chesapeake Bay watershed, labeling each pixel as either forest or non-forest landcover, as an application of the support vector machine (SVM) model approach to linear classification.

Motivation: Climate Change

I am very interested in applications of data mining to problems of sustainability and climate change. With the rapid increase in high-resolution satellite image data, computing power, and accesible machine learning software tools, there are many new and interesting ways to analyze the composition of our physical landscape from a "bird's eye" view, including detecting methane leaks, tracking deforestation, estimating carbon sequestration, and many more [4].

A high-level problem: Tree cover and the urban heat island effect

I was initally interested in exploring the relationship between tree cover (or lack thereof) and the urban heat island (UHI) effect--the phenomenon of increased temperature in urban areas compared to surrounding rural areas primarily due to human modification of the earth's surface. [5]



The urban heat island effect temperature profile [10]

A key lower-level sub-problem: Landcover classification

To explore the relationship between the level of tree cover and urban/rural temperature difference, we must be able to quantify the amount of tree cover in a given area.

In this project, I explore this key sub-problem.

BACKGROUND KNOWLEDGE

Support Vector Machines (SVMs)

Support Vector Machines are one of many linear-model-based approaches to classification. Though SVMs are extensible to multi-class ($K > 2$) classification, and even regression, we will

limit our discussion to the simpler binary classification problem.

$\vec{x}_t$ is a feature vector in d dimensional space, and r_t is the label value representing the class to which $\vec{x}_t$ belongs. Our training set can then be expressed as
$X = \left\{ (\vec{x}_t, r_t) \text{ pairs } | \vec{x}_t \in \mathbb{R}^d, r_t \in \{-1, +1\} \right\}$. [6]

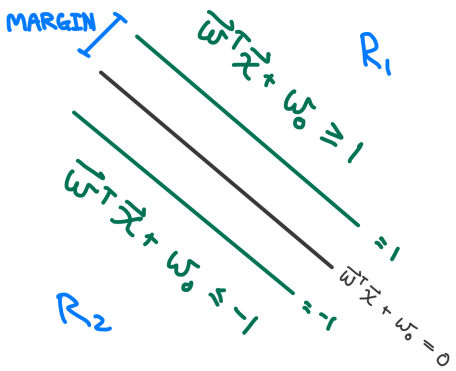
As with linear discriminant analysis, training an SVM binary classifier involves learning the weights $\vec{w}$ and intercept w_0 that define a linear discriminant hyperplane
$g(\vec{x}_t) = \vec{w}^T \vec{x}_t + w_0 = 0$, which splits the 2 classes into separate regions in feature space. However, SVM differs in a few important ways:

The Margin and Support Vectors

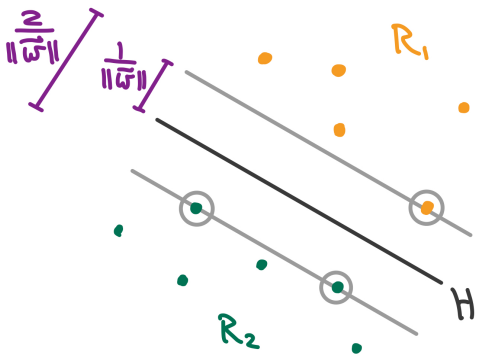
SVM's discriminant definition includes a **margin** around the discriminant hyperplane, which acts as a buffer zone to reduce mis-classification errors when random noise shifts points near one side of the discrimant onto the other. The margin is the distance between the discriminant hyperplane and the closest samples on each side.

The margin defined by setting the following condition on $\vec{w}^T$ and w_0:

$r_t(\vec{w}^T \vec{x}_t + w_0) \geq +1$



One approach to define an optimal discriminant hyperplane H is to **maximize the margin**. The process for doing so is described in Alpaydin [6] and involves multi-condition optimization using the Lagrangian, and it turns out that maximizing the margin leads to a discriminant and margin hyperplanes such that there is at least one sample from each class on their respective margins:



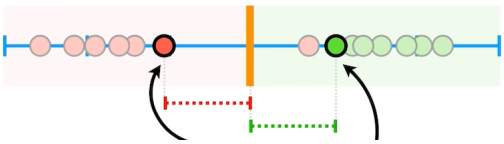
This approach allow no misclassifications of the training data.

Another, more practical, approach to defining an optimal discriminant hyperplane and its margins is **soft margins**. This solves 2 problems encountered by maximal margins:

1. The two classes may not actually be linearly separable.
2. Sometimes a training misclassification might be necessary to reduce the true error. A great illustration of this comes from the StatQuest video lecture on SVMs [9]

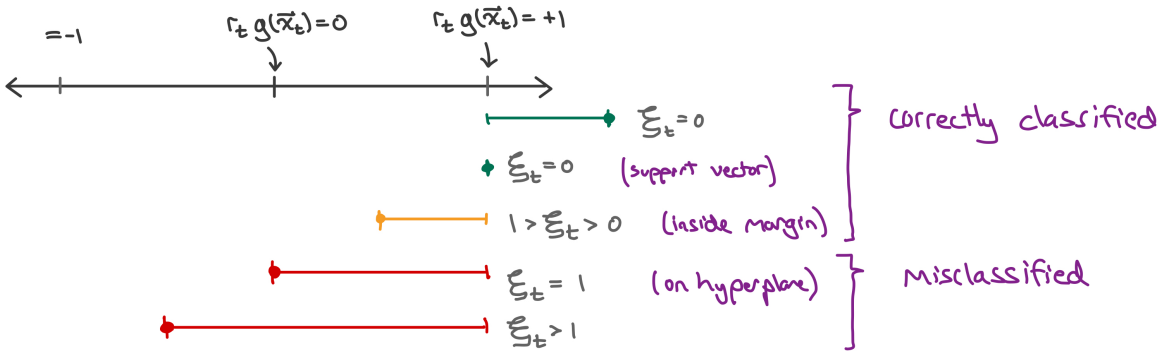
   Above we see to clear separated clusters, and the red cluster has an outlier close to the green cluster. The maximal margin approach calculates a discriminant hyperplane that is very close to the green cluster - an obvious risk for error on new data points.

   Instead, the soft margin approach intentionally misclassifies the red outlier during training, and comes up with a discriminant that is more evenly between the two clusters. This is a clear example of the **bias variance tradeoff**. We sacrifice training accuracy in exchange for lower variance.

Formally, this soft margin approach is incorporated into our optimization problem to find $\vec{w}$ and w_0 by defining the deviation of a given training point into the margin or onto the wrong side of the discriminant as ξ_t. Then we can define our margin as:

$$r_t(\vec{w}^T\vec{x}_t + w_0) \geq +1 - \xi_t$$



And we adjust our optimization to add a penalty for the overall soft error $\sum_{t=1}^{N} \xi_t$

The subset of training data points that intersect the margin hyperplanes, or fall between them are known as **support vectors**. And this subset fully defines $\vec{w}$ and w_0.

Another great advantage of the SVM model is the ability to transform the original feature space, where the 2 classes may not be linearly separable, into a higher dimension where they *are* linearly separable. There is also a famous computational optimization known as the **kernel trick** where this higher dimensional linear separability can be achieved without actually performing the (potentially expensive) transformation of the data points to the new feature space. I will not go into the details of these transformations as SVM with a *linear kernel* is sufficient for the classification I performed.

Image Segmentation

Image segmentation is one important application of classification. Given an image represented as a 2D or 1D array of pixels, each pixel is labeled with a class C_i. This form of image segmentation is specifically known as **semantic segmentation**. [7] We will apply SVM model classification to perform segmentation on our satellite images, classifying each pixel (representing a square meter of land) as either forest landcover or non-forest landcover.

Naive Implementation of SVM for segmentation

A simple naive representation of an image could be in a 1-dimensional feature space, with the only feature being color. Each x_i is color value. For simplicity let's assume the 'grayscale' scalar encoding of color (This simplification admittedly loses information from a 3-dimensional RGB representation of color, but the concepts that we'll explore next apply in either case).

So we have our training set of x_is and y_is, and can apply the SVM approach to learn the weights $\vec{w}$ and w_0 for our linear classifier.

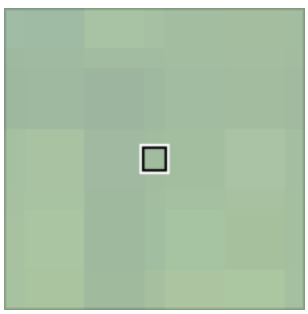
Problems with the naive implementation

This naive implementation has a problem: Color alone may not be a sufficiently informative dimension (especially grayscale, but also RGB). Intuitively, even if we assume a specific time of year and geographic location to remove the seasonality of forest leaf coverage, there's no guarantee that other types of landcover are a different color than a forest. Perhaps the grass of a field, or a roof painted green, are a close color to a nearby forest. In this case the feature vectors may not be separable and our classifier would be ineffective.

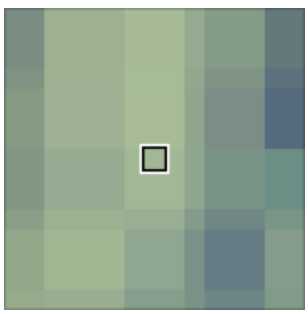
A more sophisticated implementation: Pre-processing and Feature Extraction

Since color alone may not be enough information to classify a given pixel, we can compute additional features of the pixel from its local surroundings (i.e. neighborhood [7]), including **texture** or closeness to **edges**. This is achieved by pre-processing the data and is known as **feature extraction**. [6]

A good illustration would be the texture surrounding a given pixel:



Field Texture



Forest Texture

Above are two images sampled from the data set, one a small area from a field and the other from a forest. Though the central pixels have a close color value, we see the 'texture', or the spatial arrangement of the colors and intensities in the neighborhood around the central pixels are quite different. The field is a relatively homogeneous green, while the forest has more variety.

In general this computation of new features that describe the relationship of a pixel to its surrounding neighborhood is achieved by applying a **convolutional mask** [7] to each pixel that gives it a value that is a function of the surrounding pixels.

METHODOLOGY

Data

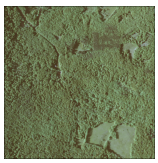
The data is comprised of high resolution aerial imagery of the Chesapeake Bay watershed [2]. The geographic area is divided into fixed size *tiles* where each pixel represents approximately 1 square meter. Every pixel of a given tile of land is labeled according to its landcover category (water, forest, field, impervious surface, etc.). The labels of a given tile are provided as images of the same dimension, where each pixel's value is $[r_t, r_t, r_t]$ (so it is RGB compatible) where r_t is the landcover label value.

Below are previews of the images and their labels used in this analysis (*Note: the labels shown here were pre-processed from the original 6 categories to 2 categories. White represents forest, and black represents non-forest*). Image 1 and its labels are the training set, and the remaining three images form the validation set.

Img 1
(training)

Img 2

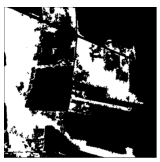
Img 3



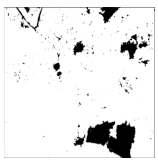
Img 4

Img 1
Labels

Img 2
Labels



Img 3
Labels



Img 4
Labels

Features, Labels, and Sample Set

The pixels from our first image serve as our training set, and the grayscale (scalar) color values of each pixel serve as the first feature.

I then construct a binary classifier, with the two labels representing either non-forest or forest landcover by mapping the provided 6 landcover categories to 2 during the initial pre-processing.

The initial sample set can be understood as follows:

$$T_0 = \{x_t, y_t | x_t \in [0, 255], y_t \in \{0, 1\}\}$$

This is only the 'initial' training set, because the next step is to perform feature extraction to compute additional numeric features (producing $d = 37$ total features) to encode texture and

edge information that will improve our ability to distinguish between the classes. That will produce the sample set:

$$T = \left\{ \vec{x}_t, y_t | \vec{x}_t \in \mathbb{R}^d, y_t \in \{0, 1\} \right\}$$

Where $t = 0, \cdots, N$ for N the number of pixels (samples) in the flattened pixel array representing the image. This training set T is what we will input into our SVM learner.

Pre-processing and feature extraction

I pre-process the data set for 2 main reasons:

1) Size/Complexity reduction

Reducing the size of the data set makes the run-time on a laptop bearable, and simplifying the problem complexity aids in a more succinct presentation. The following pre-processing steps are performed:

- Reduce the size of the sample set: The full tile image is huge (e.g. 7577x5772 pixels). I reduce this to a 1024x1024 pixel sub-tile taken from the upper left corner of the original image.
- Convert the pixel representations from RGB 3-tuples to scalar grayscale values.
- Reduce the number of labels from 6 to 2 to simplify this into a binary classification question: Is this pixel forest landcover or not?
- Map the land cover labels $\in \{0, 1\}$ (not visually distinguishable from black) to distinguishable grayscale colors $\in \{0, 255\}$

2) Feature Extraction: Texture and edges.

I apply feature extraction filters to generate new features that contain additional useful information to classify each pixel: Texture and edge context.

There are many methods for extracting texture and edge features. The methods I apply in this project were Gabor, Gaussian, and Median filters to extract texture information, and the Canny method of edge extraction. All of these extraction techniques are explained by Dr. Sreenivas Bhattiprolu in his image segmentation tutorial series [1] and also in [7]. I use the implementations from the `scipy ndimage` and `cv2` python packages.

Model Training

The training set T (the 1024^2 pre-processed pixels from image 1 and their accompanying binary labels) is input to the `scikitlearn` python package's implementation of SVM using a linear kernel [3]. After feature extraction, the feature space consists of $d = 37$ features, including the original grayscale color feature and 36 features features extracted via Gabor, Gaussian, Median, and Canny edge methods.

Training takes approximately 7 seconds (*Caveat*: Varies by about 0.5 seconds on different runs on my laptop) and converges after about 10 iterations. *See notebook code output below for exact metrics from a specific run.*

Measuring Accuracy and Error

I test the accuracy of the SVM classifier by performing classification on an image's pixels and comparing the predicted labels with the true labels.

Here accuracy is presented as the ratio of $\frac{\text{num correctly classified pixels}}{\text{num pixels}}$.

This metric applies in two important contexts:

1. The **training error or accuracy**: How well the classifier predicts the labels for the data set it was trained upon. i.e. Classify the training image.
2. The **test error or accuracy** for a given test (i.e. validation) set: How well the classifier predicts the labels for other new datasets. i.e. Classify new images. If the test set is large enough it serves as an approximation for the **true error or accuracy** of our classifier. I classify 3 images as our validation set and average their accuracy as a *very* rough approximation of the true accuracy.

When the training and test/validation accuracy are different we get a glimpse into where on the **bias-variance trade off** spectrum our classifier lies.

Empirical Error

More formally, the accuracy can be expressed in terms of error. For example the training accuracy can be expressed in terms of the *empirical error* L_s as:

$$\text{accuracy} = 1 - L_s = 1 - \frac{\{i \in [1, N] \mid \hat{y}_t \neq y_t\}}{N}$$

Where N is the number of training samples, and $\hat{y}_t$ and y_t are the predicted and actual label values for sample x_t, respectively.

RESULTS

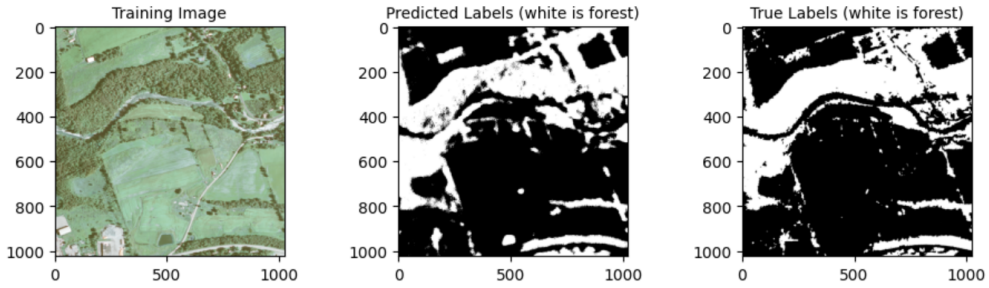
The SVM model achieves a training accuracy of 89%, and the average validation accuracy is 90%. Here is the output from a run of our classification accuracy assessment:

```
[SVM] Training set accuracy for img1: 0.89
[SVM] Test set accuracy for: img2:0.89. img3:0.89. img4:0.94.
[SVM] Average test set accuracy: 0.90
```

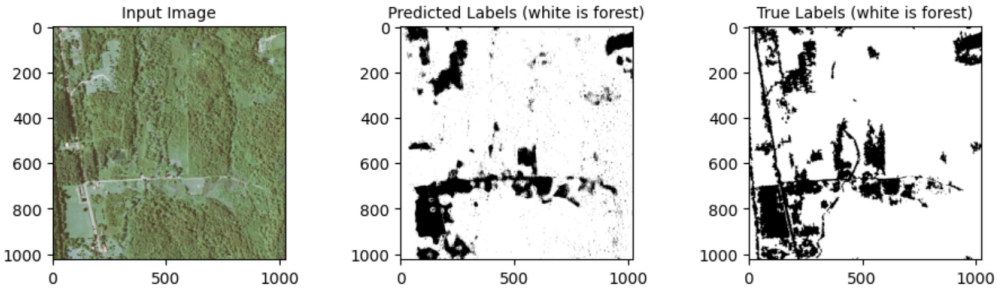
Visualizing our results

Below is a visual comparison between the original images, their predicted labels, and the actual labels. As expected due to the high accuracy scores, the predicted labels look very close to the true labels.

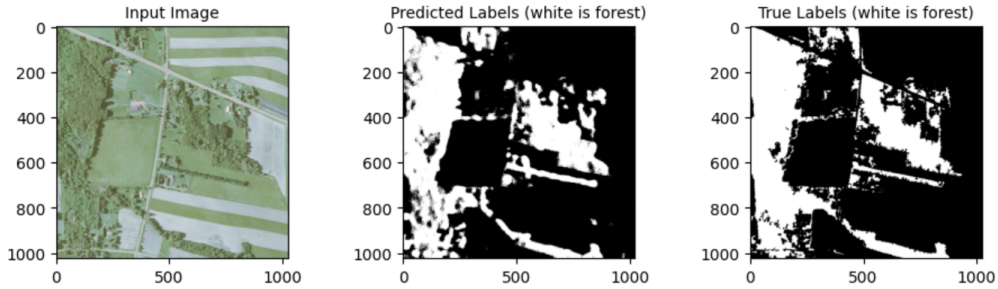
SVM image 1 (with extracted features: median,edge,gabor,gaussian)



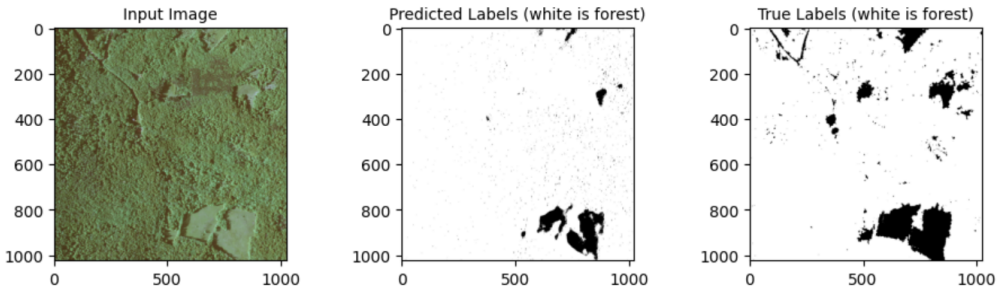
SVM image 2 (with extracted features: edge,gabor,median,gaussian)



SVM image 3 (with extracted features: edge,gabor,median,gaussian)



SVM image 4 (with extracted features: edge,gaussian,gabor,median)



Analysis

Alternative approaches

Linear Discriminant Analysis

The fact that the SVM approach with a simple linear kernal produces these good results implies that our features are linearly separable in the original feature space. There is no need to take advantage of SVM's trick of transforming the original feature space into a new higher-dimensional feature space to obtain that linear separability.

That implies other classification approaches, such as linear discriminant analysis, will also work on the un-transformed feature space.

To demonstrate this, I train a Linear Discriminant Analysis based model on the same data set and produce results of comparable accuracy to SVM:

```
[LDA] Training set accuracy for img1: 0.89
[LDA] Test set accuracy for: img2:0.89. img3:0.89. img4:0.94.
[LDA] Average test set accuracy: 0.91
```

See the notebook code below for full details.

Convolutional Neural Networks (CNNs)

SVMs and Linear Discriminant Analysis are considered 'traditional' machine learning approaches to image segmentation [1], but another common 'deep learning' approach to image classification and segmentation is Convolutional Neural Networks (CNNs) [8] One major advantage of that approach is one does not need to manually specify the filters to apply during feature extraction, which requires domain knowledge and assumptions, but instead can allow the CNN to learn those as well.

Which features are important?

In the SVM model approach, one can intuit that extracted texture and edge features would improve our classifier's ability to discriminate between the classes, and I include them. But none of the following questions have been addressed yet:

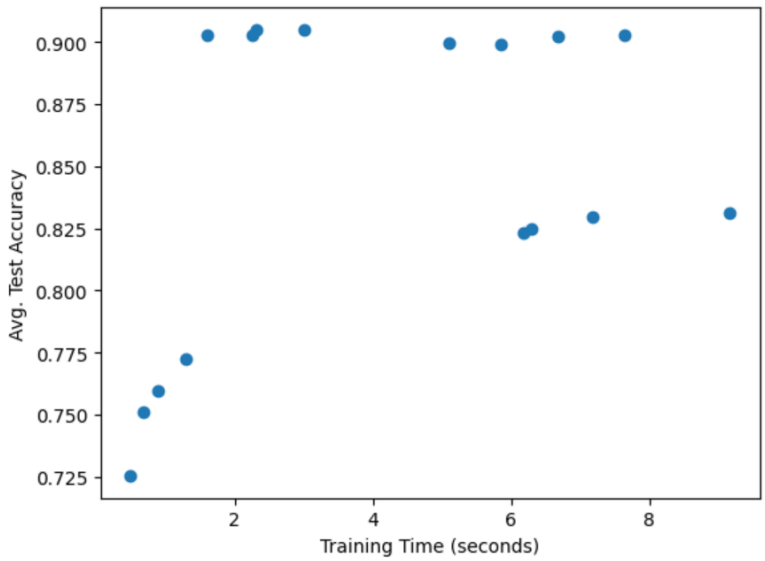
- What are the costs of including these extracted features?
- Which extracted features are most useful?
- What if some or all of the extracted features are omitted?

In an attempt to answer some of these, I train an SVM model for each possible combination of extracted feature sets in the powerset of { Gabor, Gaussian, Median, Edge } and compare their accuracy and training time. *See the notebook code below for details*.

As expected, adding in more extracted features generally increases the training time (*Note that the Gabor and Gaussian extracted feature actually consist of multiple features: 32 for Gabor and 2 for Gaussian which accounts for the large jump in training time when Gabor filters are included*):

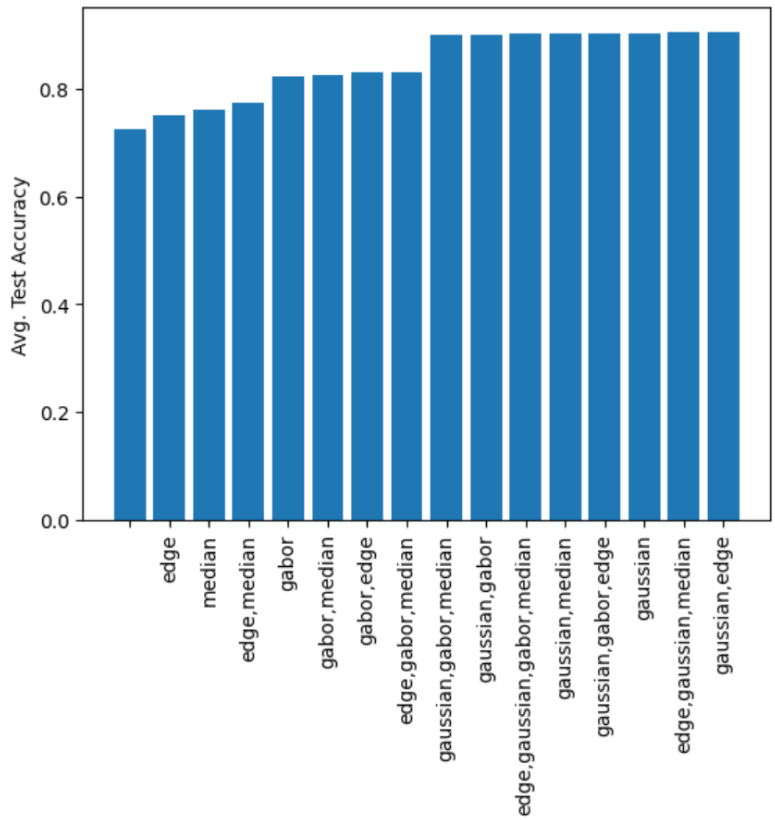
id	train_time
	0.496840
edge	0.682553
median	0.896595
edge,median	1.306012
gaussian	1.601038
gaussian,median	2.268236
gaussian,edge	2.309535
edge,gaussian,median	3.018258
gaussian,gabor	5.098367
gaussian,gabor,median	5.863657
gabor	6.171862
gabor,median	6.295949
edge,gaussian,gabor,median	6.683346
gabor,edge	7.178183
gaussian,gabor,edge	7.650589
edge,gabor,median	9.165120

But we don't always get increased accuracy in exchange for the additional computational time and effort of including more extracted features:



At the top left corner of the above graph, is a result in the lower range of training times and with the highest of accuracies. Clearly this is the most desirable approach among those considered here, and corresponds to the scenario where only Gaussian features are extracted and input (along with with the original grayscale color feature) into the model trainng.

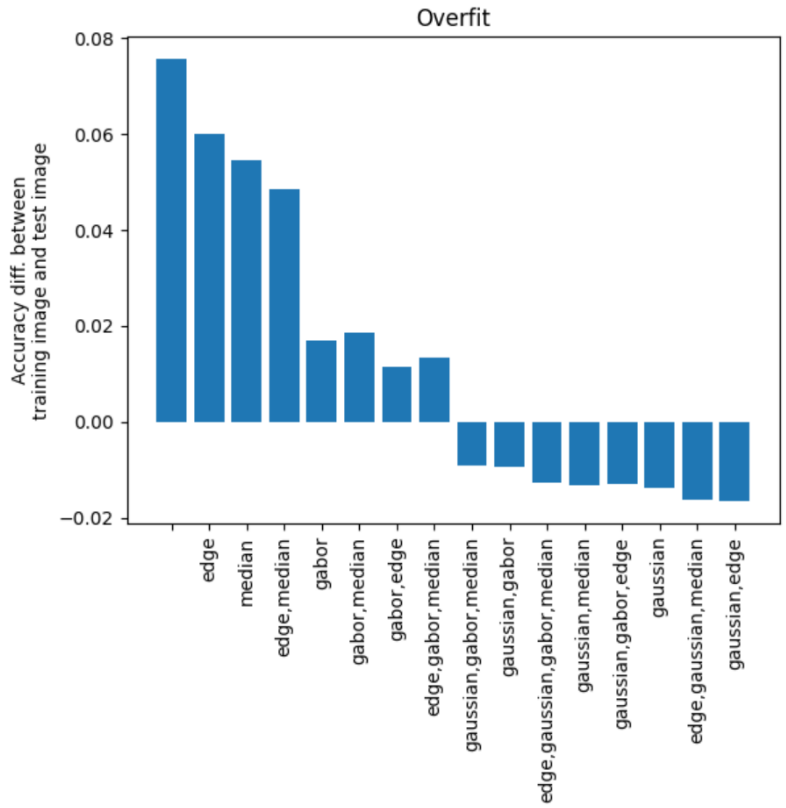
This suggests that the Gaussian filter extracted features are the most impactful. Further evidence of this is seen in the graph below, which shows the average validation accuracy of all the possible combos of extracted features. Any model trained on the Gaussian features performs well (and there is not much difference within that set).



The above graph also shows that omitting all extracted features and relying solely on the single grayscale color feature has the worst accuracy.

Bias vs. Variance Tradeoff

The graph below show an **overfit** score for all the possible combos of extracted features to train the model upon. This overfit score is simply the difference between a model's training accuracy and average test accuracy. This is a proxy for overfit, as it shows how the model may predict pixels from the training image well (every approach does better that 80% - so they all have the same relatively low bias) but classify pixels from the different test images less accurately (variance). This graph shows that the inclusion of Gaussian extracted features is also the best bet for reducing this variance. Interestingly the inclusion of Gaussian features leads to better validation accuracy than training accuracy.



CONCLUSION

One can train a fairly accurate SVM classifier to perform semantic segmentation on satellite images, classifying each square meter of the imaged area as either forest and non-forest. This approach solves the landcover classification sub-problem proposed in the introduction, and therefore could serve as a key tool to explore the urban heat island effect, given satellite images and spatial temperature data for urban areas and their local rural surroundings. Hopefully I can get to that next but it is beyond the scope of this write up!

REFERENCES

[1] Bhattiprolu, Sreenivas. *Tutorial 79 - Image segmentation using traditional Machine Learning - Part 1*. https://www.youtube.com/watch?v=uWTzkUD3V9g&list=PLHae9ggVvqPgyRQQOtENr6hK0m1UquGaG&index=81

[2] Robinson C, Hou L, Malkin K, Soobitsky R, Czawlytko J, Dilkina B, Jojic N. *Large Scale High-Resolution Land Cover Mapping with Multi-Resolution Data.* Proceedings of the 2019 Conference on Computer Vision and Pattern Recognition (CVPR 2019). https://lila.science/datasets/chesapeakelandcover

[3] scikit-learn. *sklearn.svm.LinearSVC*. https://scikit-learn.org/stable/modules/generated/sklearn.svm.LinearSVC.html

[4] Rolnick, David, Priya L. Donti, Lynn H. Kaack, Kelly Kochanski, Alexandre Lacoste, Kris Sankaran, Andrew Slavin Ross et al. *Tackling climate change with machine learning.* ACM Computing Surveys (CSUR) 55, no. 2 (2022): 1-96.

[5] Schatz J and Kucharik C J. 2015. *Urban climate effects on extreme temperatures in Madison, Wisconsin, USA*. Environmental Research Letters, Volume 10, Number 9

[6] Alpaydin, Ethem. *Introduction to Machine Learning. 2nd ed.* The MIT Press (2010): Section 13.2.

[7] Shapiro, Linda, George Stockman. *Computer Vision*. Prentice Hall (2001): Section 3.2.

[8] IBM. *Convolutional Neural Networks*. https://www.ibm.com/cloud/learn/convolutional-neural-networks

[9] StatQuest with Josh Starmer. *Support Vector Machines Part 1 (of 3): Main Ideas!!!* https://www.youtube.com/watch?v=efR1C6CvhmE

[10] Wikipedia. *Urban heat island*. https://en.wikipedia.org/wiki/Urban_heat_island#/media/File:Urban_heat_island.svg

CODE

The following executable jupyter notebook details the methodology and results informing the above writeup.

In [1]:
```
import time
import itertools
from importlib import reload
from pathlib import Path

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import cv2
from scipy import ndimage as nd
from sklearn.svm import LinearSVC
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn import metrics

import utils
reload(utils)
utils.run_tests()
```

Load the images and labels, and reduce their size:

In [2]:
```
# Return a 'subtile' of the given 2D np array representation of a tile image
# These tiles are too huge (e.g. 7577x5772 pixels), we'll need to reduce this.
# For now just grab top left subtile.
# Perhaps later I'll implement a uniform random sampling of subtiles.
def subtile(img):
    subtile_w = 1024
    subtile_h = 1024
    return img[0:subtile_h, 0:subtile_w]

# For each tile, load the image and labels as 2D numpy arrays.
# Each pixel represented by 3 values: (R, G, B)
# For the features, each pixel is a true color RGB values
# For the labels, each pixel is a 3-tuple repeating the class ID
imgs = []
```

```
labels = []
print('Loading tile image/label data into memory...')
t0 = time.time()
for i in range(len(utils.tile_asset_urls)):
    # Load NAIP image data and crop it to subtile size
    img = subtile(cv2.imread(utils.tile_asset_urls[i]['img_path']))
    imgs.append(img)
    # Load Chesapeake Conservancy land cover label data
    label_img = subtile(cv2.imread(utils.tile_asset_urls[i]['label_img_path']))
    labels.append(label_img)
    print(f'  Loaded image/label data for tile {i}')
print(f'Took {time.time() - t0:.2} seconds load data for {len(utils.tile_asset_url

# We will train on the first tile
img1_color = imgs[0]
y1_color = labels[0]
```

```
Loading tile image/label data into memory...
  Loaded image/label data for tile 0
  Loaded image/label data for tile 1
  Loaded image/label data for tile 2
  Loaded image/label data for tile 3
Took 9.7 seconds load data for 4 tiles.
```

Pre-Process our data

Perform size reduction and feature extraction pre-processing.

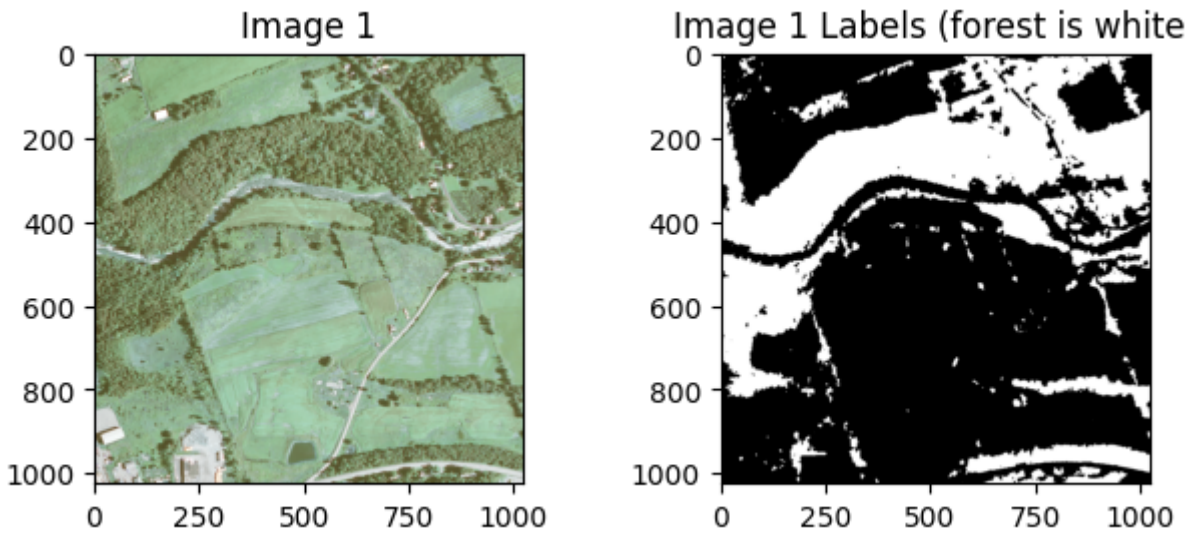
The following techniques for extracting texture and edge information around each pixel were presented by Dr. Sreenivas Bhattiprolu in his image segmentation tutorial series. [1]

In [3]:

```
# For the given grayscale image data and dataframe, return a new dataframe with
# the specified extracted feature columns.
def append_extracted_features(img, df_orig,
                              features_to_extract={'gabor', 'gaussian',
                                                   'edge', 'median'}):
    extracted_features = []
    df = df_orig.copy()

    if 'gabor' in features_to_extract:
        # Gabor features to quantify texture context
        # Here I am directly using the filter bank presented by Dr. Sreenivas
        # Bhattiprolu to generate a bank of 32 Gabor masks and apply them to
        # generate 32 new feature vectors
        num = 1
        kernels = []
        # Iterate through 32 combos of gabor params
        for theta in range(2):
            theta = theta / 4. * np.pi
            for sigma in (1, 3):  #Sigma with 1 and 3
                for lamda in np.arange(0, np.pi, np.pi / 4):
                    for gamma in (0.05, 0.5):
                        gabor_label = 'Gabor' + str(num)
                        ksize=9
                        kernel = cv2.getGaborKernel((ksize, ksize), sigma,
                                                    theta, lamda,
                                                    gamma, 0, ktype=cv2.CV_32F)
                        kernels.append(kernel)
                        # Apply the filter and append extracted feature
                        # column to the dataframe
```

```
                        fimg = cv2.filter2D(img, cv2.CV_8UC3, kernel)
                        filtered_img = fimg.reshape(-1)
                        df[gabor_label] = filtered_img
                        num += 1

    if 'gaussian' in features_to_extract:
        # Generate Gaussian filtered feature columns, also for texture
        # information
        # Gaussian with sigma=3
        gaussian_img = nd.gaussian_filter(img, sigma=3)
        gaussian_img1 = gaussian_img.reshape(-1)
        df['Gaussian s3'] = gaussian_img1

        # Gaussian with sigma=7
        gaussian_img2 = nd.gaussian_filter(img, sigma=7)
        gaussian_img3 = gaussian_img2.reshape(-1)
        df['Gaussian s7'] = gaussian_img3

    if 'median' in features_to_extract:
        # MEDIAN with sigma=3
        median_img = nd.median_filter(img, size=3)
        median_img1 = median_img.reshape(-1)
        df['Median s3'] = median_img1

    if 'edge' in features_to_extract:
        # Edge information
        edges = cv2.Canny(img, 100, 200)
        edges1 = edges.reshape(-1)
        df['Canny Edge'] = edges1 #Add column to original dataframe

    return df

# Returns a dataframe with the original color images' grayscale feature
# column and the specified extracted feature columns
def preprocess_features(img_color,
                        features_to_extract={'gabor', 'gaussian',
                                             'edge', 'median'}):
    df = pd.DataFrame()

    # Convert to grayscale to reduce dimensionality
    # Each pixel goes from a 3-value RGB list to a single scalar
    # value in [0,256)
    img_gray = cv2.cvtColor(img_color, cv2.COLOR_BGR2GRAY)

    # Flatten the 2D image and label data into 1D columns
    df['grayscale'] = img_gray.reshape(-1)
    df = append_extracted_features(img_gray, df,
                                   features_to_extract=features_to_extract)
    return df

# Returns a 2 tuple consisting of the following preprocessed versions of
# the given 2D RGB label array:
# 1) The 2-class, visible label image in grayscale RGB.
#    i.e. color RGB -> gray RGB
# 2) The flattened scalar label array.
#    i.e. 2D color RGB -> 2D gray RGB -> 2D scalar -> 1D scalar
def preprocess_labels_binary(y_color):
    # Map the labels to a binary set: tree = 1, non-trees = 0
    y_color = np.where(y_color == 2, np.uint8(1), np.uint8(0))
    y_gray = utils.bin_to_viz(y_color) # 2D RGB visible gray
    y_gray_scalar = cv2.cvtColor(y_gray, cv2.COLOR_BGR2GRAY)
    y = y_gray_scalar.reshape(-1)
    return (y_gray, y)
```

```
X1_df = preprocess_features(img1_color)
y1_gray, y1 = preprocess_labels_binary(y1_color)

# Visualize
fig, axs = plt.subplots(nrows=1, ncols=2, figsize=(6,6))
fig.tight_layout(h_pad=2, w_pad=4)
axs[0].imshow(img1_color)
axs[0].set_title('Image 1')
axs[1].imshow(y1_gray)
axs[1].set_title('Image 1 Labels (forest is white)')
plt.show()
```



Above we see the original sub-tile color image and its labels, which classify each pixel (1 square meter area) as either forest landcover or not.

Train our model

Run the SVM algorithm to learn the parameters that define the optimal linear discriminant hyperplane. We use image 1 as our training data set.

In [4]:
```
def fit_model(X_train, y_train):
    print(f'Training the Linear SVM model on the data. N={len(X_train)} samples. d

    # Fit the data with a SVM model using a linear kernel
    # Since N>d, perform primal rather than dual optimization
    clf = LinearSVC(verbose=False, dual=False, max_iter=300)

    t0 = time.time()
    clf.fit(X_train, y_train)
    print(f'Training took {time.time() - t0:.2} seconds. {clf.n_iter_} iterations

    return clf

clf = fit_model(X1_df, y1)

print(f'Learned weight vector w: {clf.coef_[0]}')
print(f'Learned intercept w_0: {clf.intercept_}')
```

```
Training the Linear SVM model on the data. N=1048576 samples. d=37 features.
Training took 6.7 seconds. 10 iterations until convergence.
Learned weight vector w: [-0.00167554  0.          0.          0.00275756  0.01001
941 -0.00246067
 -0.00098282  0.0010704   0.00228074  0.          0.         -0.00255375
  0.00091193 -0.00087646 -0.00086756 -0.00044847 -0.00037446  0.
  0.          0.00318342  0.00318342 -0.0022301   0.00767571  0.00207957
 -0.00132073  0.          0.          0.00166209 -0.00126544  0.00503936
 -0.00502832 -0.00273442  0.00356975  0.01054327 -0.03853163 -0.00313121
  0.00068078]
Learned intercept w_0: [1.24840142e-05]
```

Testing the accuracy of the SVM model

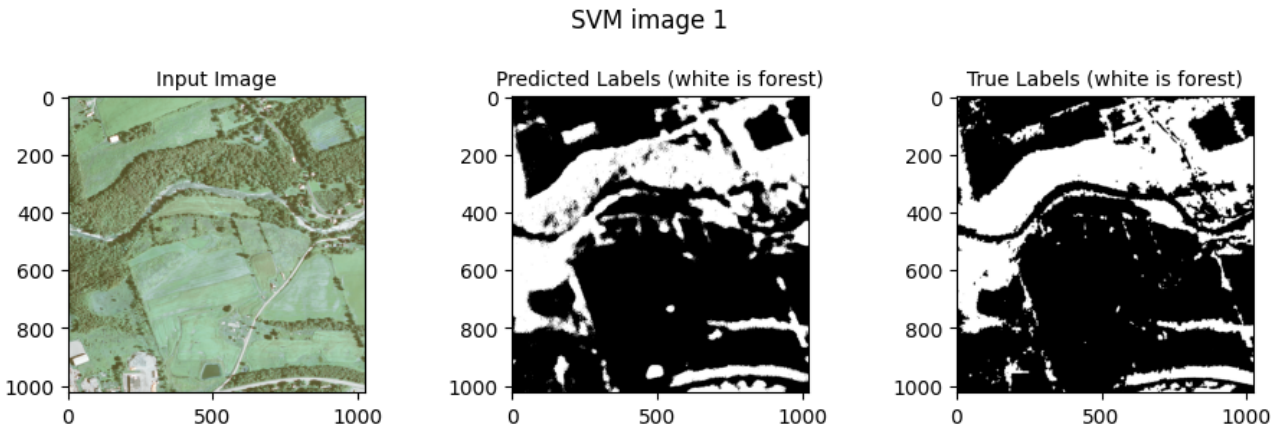
In [5]:

```python
# Additional images and their accompanying labels
img2_color, img3_color, img4_color = imgs[1], imgs[2], imgs[3]
y2_color, y3_color, y4_color = labels[1], labels[2], labels[3]

# Preprocess them
X2_df = preprocess_features(img2_color)
X3_df = preprocess_features(img3_color)
X4_df = preprocess_features(img4_color)
y2_gray, y2 = preprocess_labels_binary(y2_color)
y3_gray, y3 = preprocess_labels_binary(y3_color)
y4_gray, y4 = preprocess_labels_binary(y4_color)

# Predict the training set so we can calulate the empirical error
y1_hat = clf.predict(X1_df)

# Predict separate test sets so we can average as a rough proxy for true error.
y2_hat = clf.predict(X2_df)
y3_hat = clf.predict(X3_df)
y4_hat = clf.predict(X4_df)

# Return the accuracy ratio calculated from the pixel-by-pixel comparison
# of a given prediction to the true labels. Arguments are assumed to be
# grayed and flattened.
def accuracy(y_hat, y):
    assert len(y_hat) == len(y)
    n_pixels = len(y_hat)
    misclassified = 0
    for i in range(len(y_hat)): # flat, grayscale labels (mapped to 0,255)
        if y_hat[i] != y[i]:
            misclassified += 1
    return 1.0 - misclassified/n_pixels

img1_accuracy = accuracy(y1_hat, y1) # Training accuracy
img2_accuracy = accuracy(y2_hat, y2)
img3_accuracy = accuracy(y3_hat, y3)
img4_accuracy = accuracy(y4_hat, y4)
avg_test_accuracy = (img2_accuracy + img3_accuracy + img4_accuracy) / 3
print(f'[SVM] Training set accuracy for img1: {img1_accuracy:.02f}')
print(f'[SVM] Test set accuracy for: img2:{img2_accuracy:.02f}. img3:{img3_accurac
print(f'[SVM] Average test set accuracy: {avg_test_accuracy:.02f}')
```

```
[SVM] Training set accuracy for img1: 0.89
[SVM] Test set accuracy for: img2:0.89. img3:0.89. img4:0.94.
[SVM] Average test set accuracy: 0.90
```

Visualizing Empirical Training Accuracy

In [6]:
```
# Visualize
def visual_comparison(orig_img_color, y_hat_gray, y_gray, title='SVM Model'):
    fig, axs = plt.subplots(nrows=1, ncols=3, figsize=(9,9))
    fig.suptitle(title, fontsize='large')
    fig.tight_layout(h_pad=2, w_pad=4)
    plt.subplots_adjust(top=1.55)
    axs[0].imshow(orig_img_color)
    axs[0].set_title('Input Image', fontsize='medium')
    axs[1].imshow(y_hat_gray)
    axs[1].set_title('Predicted Labels (white is forest)', fontsize='medium')
    axs[2].imshow(y_gray)
    axs[2].set_title('True Labels (white is forest)', fontsize='medium')
    plt.show()

# Convert back to a 2D RGB grayscale representation
y1_hat_gray = cv2.cvtColor(y1_hat, cv2.COLOR_GRAY2RGB).reshape(img1_color.shape)
visual_comparison(img1_color, y1_hat_gray, y1_gray, title=f'SVM image 1')
```



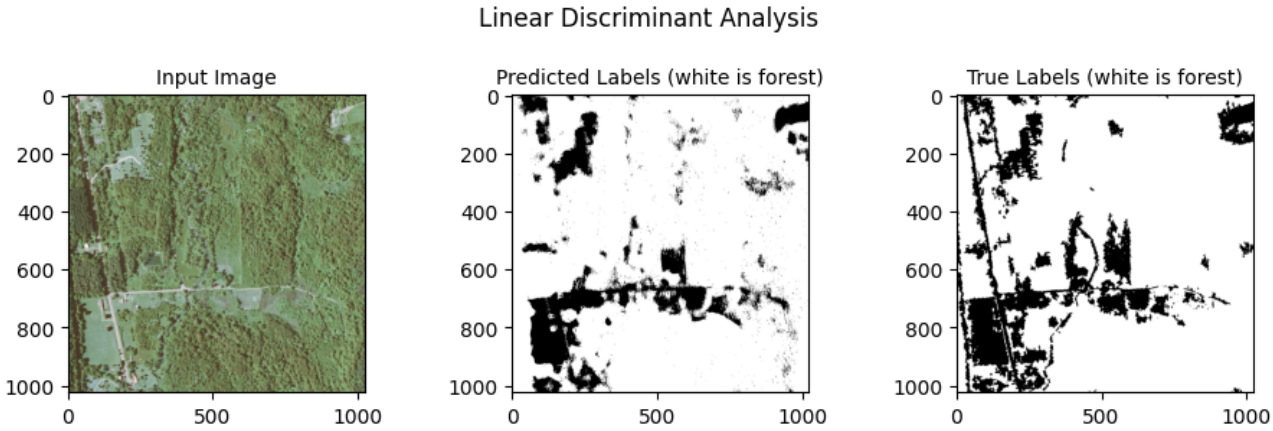
Above we see a visual representation of the difference between the predicted and true labels for each pixel. As we would expect from the high accuracy, the predicted labeling is visually quite close to the actual labels.

Applying another classification algorithm: Linear Discriminant Analysis

In [7]:
```
ld_clf = LinearDiscriminantAnalysis()
t0 = time.time()
ld_clf.fit(X1_df, y1)
print(f'Training took {time.time() - t0:.2} seconds.')

y1_hat_ld = ld_clf.predict(X1_df)
y2_hat_ld = ld_clf.predict(X2_df)
y3_hat_ld = ld_clf.predict(X3_df)
y4_hat_ld = ld_clf.predict(X4_df)

y2_hat_ld_gray = cv2.cvtColor(y2_hat_ld, cv2.COLOR_GRAY2RGB).reshape(img2_color.sh
visual_comparison(img2_color, y2_hat_ld_gray, y2_gray,
                  title='Linear Discriminant Analysis')

ld_accuracy = accuracy(y2_hat_ld, y2)

training_accuracy_ld = accuracy(y1_hat_ld, y1)
img2_accuracy_ld = accuracy(y2_hat_ld, y2)
img3_accuracy_ld = accuracy(y3_hat_ld, y3)
img4_accuracy_ld = accuracy(y4_hat_ld, y4)
```

```
avg_test_accuracy = (img2_accuracy_ld + img3_accuracy_ld + img4_accuracy_ld) / 3

print(f'[LDA] Training set accuracy for img1: {training_accuracy_ld:.02f}')
print(f'[LDA] Test set accuracy for: img2:{img2_accuracy_ld:.02f}. img3:{img3_accu
print(f'[LDA] Average test set accuracy: {avg_test_accuracy:.02f}')
```

```
Training took 5.4 seconds.
```




```
[LDA] Training set accuracy for img1: 0.89
[LDA] Test set accuracy for: img2:0.89. img3:0.89. img4:0.94.
[LDA] Average test set accuracy: 0.91
```

We see that the Linear Discriminant analysis approach works with the same accuracy as the SVM approach.

Investigating the importance of extracted features

Let's test training our SVM classifier with different sets of features and compare the accuracy and speed.

In [8]:
```
# Additional images and their accompanying labels
img3_color, img4_color = imgs[2], imgs[3]
y3_color, y4_color = labels[2], labels[3]

# Same preprocessed labels will be used in each iteration
y3_gray, y3 = preprocess_labels_binary(y3_color)
y4_gray, y4 = preprocess_labels_binary(y4_color)

features_to_extract={'gabor', 'gaussian', 'edge', 'median'}

results_df = pd.DataFrame()

ps = utils.powerset(features_to_extract)
ps.sort(key=len)
for fs in ps:
    feature_set = set(fs)
    print(f'\nRunning SVM with the following extracted features: {",".join(feature

    # Generate the specified extracted features for the training image
    X_df = preprocess_features(img1_color, features_to_extract=feature_set)

    # Train
    t0 = time.time()
    clf = fit_model(X_df, y1)
    train_time = time.time() - t0
```

```
    # Test on the first image's test set
    y_hat = clf.predict(X_df)
    training_accuracy = metrics.accuracy_score(y1, y_hat)

    # We'll test on 3 additional images
    X2_df = preprocess_features(img2_color, features_to_extract=feature_set)
    X3_df = preprocess_features(img3_color, features_to_extract=feature_set)
    X4_df = preprocess_features(img4_color, features_to_extract=feature_set)

    # Predict
    y2_hat = clf.predict(X2_df)
    y3_hat = clf.predict(X3_df)
    y4_hat = clf.predict(X4_df)
    y2_hat_gray = cv2.cvtColor(y2_hat, cv2.COLOR_GRAY2RGB).reshape(img2_color.shap
    y3_hat_gray = cv2.cvtColor(y3_hat, cv2.COLOR_GRAY2RGB).reshape(img3_color.shap
    y4_hat_gray = cv2.cvtColor(y4_hat, cv2.COLOR_GRAY2RGB).reshape(img4_color.shap
    img2_accuracy = accuracy(y2_hat, y2)
    img3_accuracy = accuracy(y3_hat, y3)
    img4_accuracy = accuracy(y4_hat, y4)
    avg_test_accuracy = (img2_accuracy + img3_accuracy + img4_accuracy) / 3

    # Visualize validation sets
    # visual_comparison(img2_color, y2_hat_gray, y2_gray,
    #                   title=f'SVM image 2 (with extracted features: {",".join(fe
    # visual_comparison(img3_color, y3_hat_gray, y3_gray,
    #                   title=f'SVM image 3 (with extracted features: {",".join(fe
    # visual_comparison(img4_color, y4_hat_gray, y4_gray,
    #                   title=f'SVM image 4 (with extracted features: {",".join(fe

    # Output stats
    print(f'[SVM] Training set accuracy for img1: {training_accuracy:.02f}')
    print(f'[SVM] Test set accuracy for: img2:{img2_accuracy:.02f}. img3:{img3_acc
    print(f'[SVM] Average test set accuracy: {avg_test_accuracy:.02f}')

    # Record relevant performance results
    perf_result = {
        'id': ",".join(feature_set),
        'train_time': train_time,
        'training_accuracy': training_accuracy,
        'img2_accuracy': img2_accuracy,
        'img3_accuracy': img3_accuracy,
        'img4_accuracy': img4_accuracy,
        'test_avg_accuracy': avg_test_accuracy,

        # As a proxy for how much the model is overfit to the training data,
        # record the difference between the accuracy of predicting pixels
        # from the training image to average accuracy of predicting test
        # pixels from the other images.
        'overfit': training_accuracy - avg_test_accuracy
    }
    # Encode which features were on/off on this run
    perf_result.update({ f:(f in feature_set) for f in features_to_extract })
    results_df = results_df.append(perf_result, ignore_index=True)
```

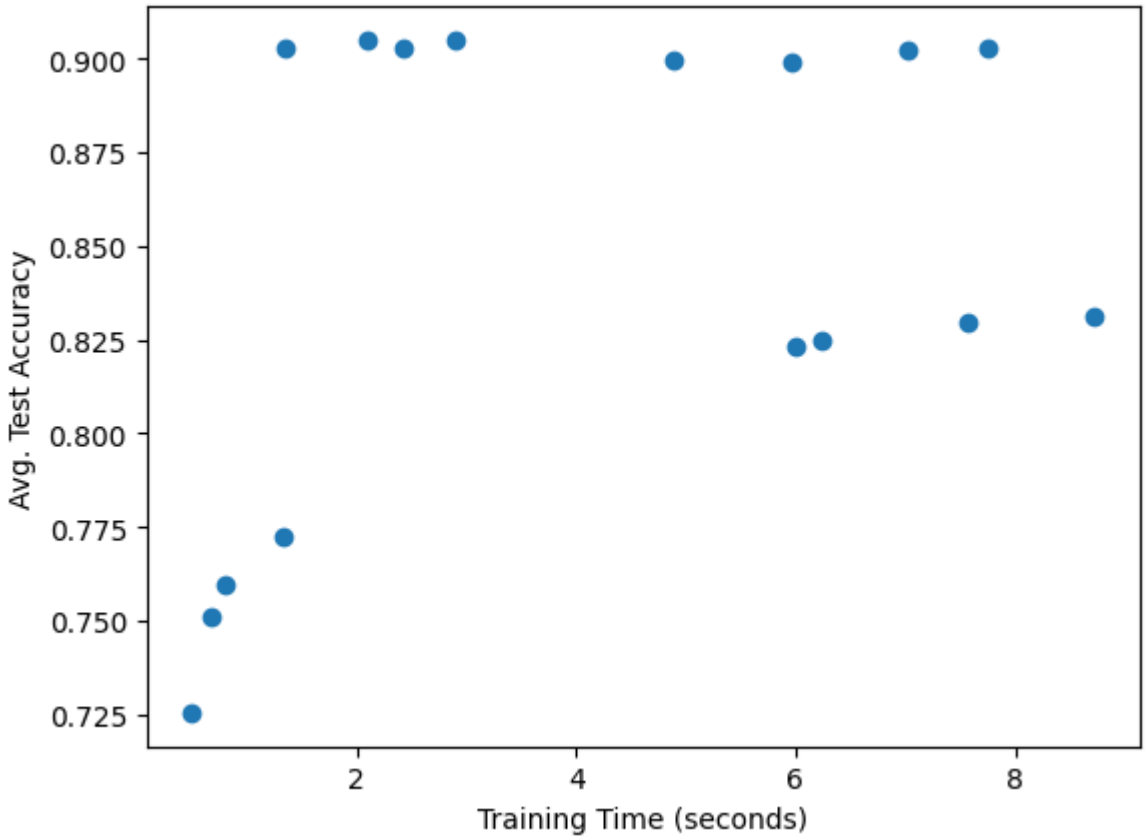
```
Running SVM with the following extracted features:
Training the Linear SVM model on the data. N=1048576 samples. d=1 features.
Training took 0.5 seconds. 10 iterations until convergence.
[SVM] Training set accuracy for img1: 0.80
[SVM] Test set accuracy for: img2:0.65. img3:0.80. img4:0.73.
[SVM] Average test set accuracy: 0.73

Running SVM with the following extracted features: gabor
Training the Linear SVM model on the data. N=1048576 samples. d=33 features.
Training took 6.0 seconds. 8 iterations until convergence.
[SVM] Training set accuracy for img1: 0.84
[SVM] Test set accuracy for: img2:0.77. img3:0.84. img4:0.86.
[SVM] Average test set accuracy: 0.82

Running SVM with the following extracted features: gaussian
Training the Linear SVM model on the data. N=1048576 samples. d=3 features.
Training took 1.3 seconds. 18 iterations until convergence.
[SVM] Training set accuracy for img1: 0.89
[SVM] Test set accuracy for: img2:0.89. img3:0.88. img4:0.94.
[SVM] Average test set accuracy: 0.90

Running SVM with the following extracted features: edge
Training the Linear SVM model on the data. N=1048576 samples. d=2 features.
Training took 0.67 seconds. 9 iterations until convergence.
[SVM] Training set accuracy for img1: 0.81
[SVM] Test set accuracy for: img2:0.67. img3:0.82. img4:0.76.
[SVM] Average test set accuracy: 0.75

Running SVM with the following extracted features: median
Training the Linear SVM model on the data. N=1048576 samples. d=2 features.
Training took 0.8 seconds. 13 iterations until convergence.
[SVM] Training set accuracy for img1: 0.81
[SVM] Test set accuracy for: img2:0.68. img3:0.81. img4:0.79.
[SVM] Average test set accuracy: 0.76

Running SVM with the following extracted features: gaussian,median
Training the Linear SVM model on the data. N=1048576 samples. d=4 features.
Training took 2.4 seconds. 24 iterations until convergence.
[SVM] Training set accuracy for img1: 0.89
[SVM] Test set accuracy for: img2:0.89. img3:0.88. img4:0.94.
[SVM] Average test set accuracy: 0.90

Running SVM with the following extracted features: gaussian,gabor
Training the Linear SVM model on the data. N=1048576 samples. d=35 features.
Training took 4.9 seconds. 8 iterations until convergence.
[SVM] Training set accuracy for img1: 0.89
[SVM] Test set accuracy for: img2:0.88. img3:0.88. img4:0.93.
[SVM] Average test set accuracy: 0.90

Running SVM with the following extracted features: gabor,median
Training the Linear SVM model on the data. N=1048576 samples. d=34 features.
Training took 6.2 seconds. 8 iterations until convergence.
[SVM] Training set accuracy for img1: 0.84
[SVM] Test set accuracy for: img2:0.78. img3:0.84. img4:0.86.
[SVM] Average test set accuracy: 0.82

Running SVM with the following extracted features: gaussian,edge
Training the Linear SVM model on the data. N=1048576 samples. d=4 features.
Training took 2.1 seconds. 19 iterations until convergence.
[SVM] Training set accuracy for img1: 0.89
[SVM] Test set accuracy for: img2:0.89. img3:0.88. img4:0.94.
[SVM] Average test set accuracy: 0.90
```

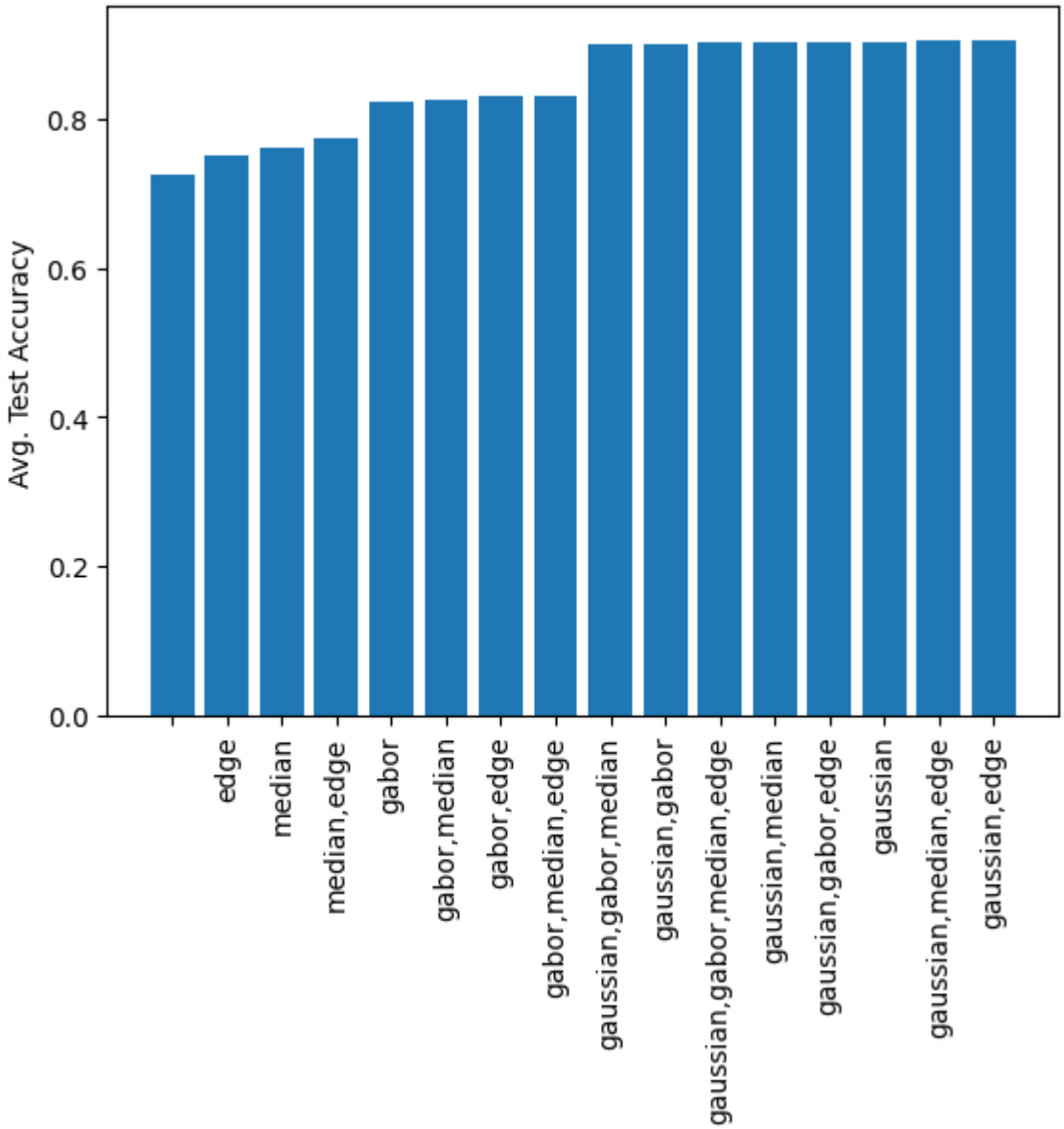
```
Running SVM with the following extracted features: median,edge
Training the Linear SVM model on the data. N=1048576 samples. d=3 features.
Training took 1.3 seconds. 15 iterations until convergence.
[SVM] Training set accuracy for img1: 0.82
[SVM] Test set accuracy for: img2:0.69. img3:0.82. img4:0.80.
[SVM] Average test set accuracy: 0.77

Running SVM with the following extracted features: gabor,edge
Training the Linear SVM model on the data. N=1048576 samples. d=34 features.
Training took 7.6 seconds. 9 iterations until convergence.
[SVM] Training set accuracy for img1: 0.84
[SVM] Test set accuracy for: img2:0.78. img3:0.84. img4:0.87.
[SVM] Average test set accuracy: 0.83

Running SVM with the following extracted features: gaussian,gabor,edge
Training the Linear SVM model on the data. N=1048576 samples. d=36 features.
Training took 7.8 seconds. 10 iterations until convergence.
[SVM] Training set accuracy for img1: 0.89
[SVM] Test set accuracy for: img2:0.89. img3:0.89. img4:0.94.
[SVM] Average test set accuracy: 0.90

Running SVM with the following extracted features: gaussian,median,edge
Training the Linear SVM model on the data. N=1048576 samples. d=5 features.
Training took 2.9 seconds. 21 iterations until convergence.
[SVM] Training set accuracy for img1: 0.89
[SVM] Test set accuracy for: img2:0.89. img3:0.88. img4:0.94.
[SVM] Average test set accuracy: 0.90

Running SVM with the following extracted features: gaussian,gabor,median
Training the Linear SVM model on the data. N=1048576 samples. d=36 features.
Training took 6.0 seconds. 9 iterations until convergence.
[SVM] Training set accuracy for img1: 0.89
[SVM] Test set accuracy for: img2:0.88. img3:0.88. img4:0.93.
[SVM] Average test set accuracy: 0.90

Running SVM with the following extracted features: gabor,median,edge
Training the Linear SVM model on the data. N=1048576 samples. d=35 features.
Training took 8.7 seconds. 10 iterations until convergence.
[SVM] Training set accuracy for img1: 0.84
[SVM] Test set accuracy for: img2:0.78. img3:0.84. img4:0.87.
[SVM] Average test set accuracy: 0.83

Running SVM with the following extracted features: gaussian,gabor,median,edge
Training the Linear SVM model on the data. N=1048576 samples. d=37 features.
Training took 7.0 seconds. 10 iterations until convergence.
[SVM] Training set accuracy for img1: 0.89
[SVM] Test set accuracy for: img2:0.89. img3:0.89. img4:0.94.
[SVM] Average test set accuracy: 0.90
```

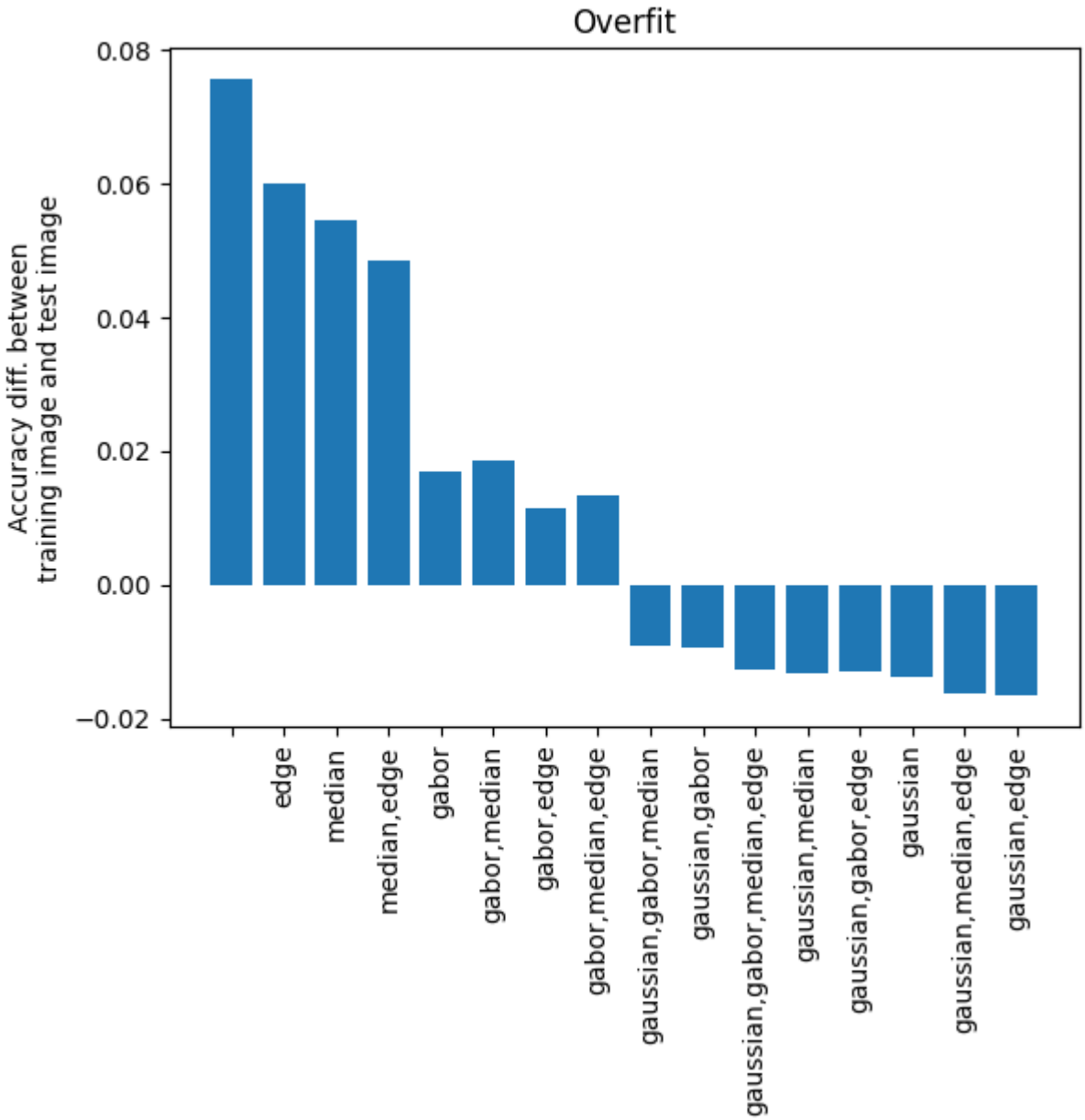
Visualizing the importance and costs of extracted features

In [9]:
```
plt.scatter(results_df.train_time, results_df.test_avg_accuracy)
plt.xlabel('Training Time (seconds)')
plt.ylabel('Avg. Test Accuracy')
plt.show()

results_df.sort_values(by=['gaussian', 'test_avg_accuracy'], inplace=True)
plt.bar(results_df.id, results_df.test_avg_accuracy)
plt.xticks(rotation='vertical')
plt.ylabel('Avg. Test Accuracy')
plt.show()
```

```
plt.bar(results_df.id, results_df.overfit)
plt.xticks(rotation='vertical')
plt.ylabel('Accuracy diff. between\ntraining image and test image')
plt.title('Overfit')
plt.show()

results_df.sort_values(by=['train_time'], inplace=True)
results_df
```




Avg. Test Accuracy
0.0
0.2
0.4
0.6
0.8
edge
median
median,edge
gabor
gabor,median
gabor,edge
gabor,median,edge
gaussian,gabor,median
gaussian,gabor
gaussian,gabor,median,edge
gaussian,median
gaussian,gabor,edge
gaussian
gaussian,median,edge
gaussian,edge


Overfit
Accuracy diff. between
training image and test image
0.08
0.06
0.04
0.02
0.00
−0.02
edge
median
median,edge
gabor
gabor,median
gabor,edge
gabor,median,edge
gaussian,gabor,median
gaussian,gabor
gaussian,gabor,median,edge
gaussian,median
gaussian,gabor,edge
gaussian
gaussian,median,edge
gaussian,edge

Out[9]:

	id	train_time	training_accuracy	img2_accuracy	img3_accuracy	img4
0		0.498266	0.801119	0.645166	0.802676	
3	edge	0.666510	0.810925	0.669106	0.819376	
4	median	0.795866	0.814092	0.680327	0.810534	
9	median,edge	1.339320	0.820977	0.692696	0.822259	
2	gaussian	1.347929	0.889219	0.889896	0.881727	
8	gaussian,edge	2.098567	0.888393	0.889226	0.884515	
5	gaussian,median	2.421692	0.889325	0.889445	0.881382	
12	gaussian,median,edge	2.898868	0.888363	0.888639	0.884266	
6	gaussian,gabor	4.896493	0.890109	0.883206	0.884191	
13	gaussian,gabor,median	5.967325	0.890181	0.883139	0.884062	
1	gabor	6.000651	0.840028	0.773067	0.839705	
7	gabor,median	6.234116	0.843536	0.775065	0.841597	
15	gaussian,gabor,median,edge	7.031569	0.889875	0.885131	0.886913	
10	gabor,edge	7.566192	0.841242	0.775925	0.843102	
11	gaussian,gabor,edge	7.759019	0.889758	0.885245	0.887087	
14	gabor,median,edge	8.723395	0.844657	0.777934	0.844597	

Utility Code

Below is pasted the utility code I wrote in a separate file and imported into this notebook:

In [ ]:

```
import itertools
from collections import defaultdict

import pprint
from pystac_client import Client

pp = pprint.PrettyPrinter(indent=2)

# Hardcoded Asset URLs
# pulled via get_tile_asset_urls()
tile_asset_urls = [
    {
        'img_path': './data/train/ny/0/naip/m_4207421_se_18_1_naip-new.tif',
        'label_img_path': './data/train/ny/0/lc/m_4207421_se_18_1_lc.tif',
    },
    {
        'img_path': './data/train/ny/1/naip/m_4207608_se_18_1_naip-new.tif',
        'label_img_path': './data/train/ny/1/lc/m_4207608_se_18_1_lc.tif',
    },
    {
        'img_path': './data/train/ny/10/naip/m_4207606_sw_18_1_naip-new.tif',
        'label_img_path': './data/train/ny/10/lc/m_4207606_sw_18_1_lc.tif',
    },
    {
        'img_path': './data/train/ny/12/naip/m_4207452_nw_18_1_naip-new.tif',
        'label_img_path': './data/train/ny/12/lc/m_4207452_nw_18_1_lc.tif',
    },
```

```
]

# Make binary classification visually distinguishable in RGB
# Maps from {0,1} to {0,255}
bin_to_viz = lambda x: x * 255

"""
    Lookup and print out the chesapeake landcover data set asset URLs from the
    STAC catalog. STAC specification is documented at https://stacspec.org.
"""
def get_tile_asset_urls(state='ny', num_items=5):
    # Config
    catalog_path_str = '/Users/davidmoench/Downloads/microsoft_chesapeake/catalog.
    collections = ['landsat_leaf_on', 'nlcd', 'naip', 'lc']

    cat = Client.open(catalog_path_str)

    # print('Collections:')
    # pp.pprint([c.id for c in cat.get_collections()])

    # Search the STAC collection for feature and label data for some tiles
    # Save the assets in the result object with the hierarchy { tile_id: { collect
    tile_data = defaultdict(lambda: {c:{} for c in collections})

    for coll_name in collections:
        coll = cat.get_collection(f'microsoft_chesapeake_{coll_name}')
        items = list(coll.get_items())
        items = list(filter(lambda x: f'_{state}_train_' in x.id, items))
        items.sort(key=lambda a: a.id)
        items = items[0:num_items]
        for item in items:
            tile_id = item.id.split("_")[-1]
            tile_data[tile_id][coll_name] = list(item.assets.values())[0].href

    # Print the href locations of the data files
    # pp.pprint(tile_data)

    # TODO use these and write a script to download the images to the ./data direc
    # in an appropriate hierarchy: ./data/train/{state}/{tile}/{collection}/{img_n
    return tile_data

"""
    Return the powerset (as a list of tuples) of the given python set.
"""
def powerset(s):
    ps = set()
    for r in range(len(s)+1):
        for combo in itertools.combinations(s, r):
            ps.add(combo)
    return list(ps)
s = {1,2,3,4}

def run_tests():
    assert len(powerset(s)) == 2 ** len(s)
```