1 Group Project: Skin Cancer Diagnostics

Class = CS 598 / STAT 542, Practical Statistical Learning

Alex Kwan, NETID = akwan2

(I chose to do this group project on my own)

12/15/2019

1.1 Project Summary

Classification, in two general thrusts, was applied to the dataset of malignant and benign skin cancer .jpg files. In one thrust, the pixel-based effort, principal components were extracted for each red, green and blue channel set of pixels. Classification models including Linear Discrimant Analysis (LDA), Random Forests (RF), and Support Vector Machine (SVM) methods, used these principal components of the pixels as the features. In another thrust, the non-pixel-based effort, new features were engineered to approximate common attributes of benign and malignant skin cancer lesions used by health providers. These features aimed to quantify the lesion symmetry and color variation. The pixel-based effort with SVM ended up providing the best accuracy. That said, the non-pixel-based effort tended to provide greater accuracy across the board for the three different classification methods.

1.2 Project Description

This project aimed to train a classifier on two sets of images (benign and malignant skin cancer lesions). A pixel-based effort as well as a non-pixel-based effort was employed. This project leaned heavily on employing the scikit-learn and scikit-image packages available with Python. These packages allowed for manipulating images and applying classification learning.

This jupyter notebook can be found in this location https://github.com/irxum/psl_proj2 (https://github.com/irxum/psl_proj2).

1.2.0.1 import packages and file paths [...]

(hiding to make report clean)

1.3 Part 1: Classification with Pixel-Based Features

1.3.1 Data Processing

To process the jpg files, I first resize each image to a standard size which I chose to be the size of the smallest image. I then take each image and split it into its red, green and blue channels. For each image channel, I get a vector of values. I do this for the benign and malignant set. I combine all image channel vectors into a large matrix for each channel. With this matrix, I use scikit-learn to get the principal components. I choose the principal components that explain 97.5% of the variation. Doing this cuts down the number of features quite a bit. For example, I ended up only using 65 principal components for the red channel. If I used each pixel, I would end up with over 300k features for each channel. With the most useful principal components from each channel, I recombine to build one matrix with all channel principal components. I then normalize for each feature and that becomes by set of features used with my classifiers.

1.3.1.1 pull in images

(hiding to make report clean)

In [3]:
```
# So convert each image to 718 x 542 by resizing
cat_ls = ['benign','malignant']
new_subdir = 'resized'
new_size = (718, 542)

# Only need to do once so this if statement is to prevent re-run
run_below = False

if run_below:
    for c in cat_ls:
        dir_fp = os.path.join(base_path, c)
        jpg_ls = [f for f in os.listdir(dir_fp) if '.jpg' in f]
        for j in jpg_ls:
            j_fp = os.path.join(dir_fp, j)
            im = Image.open(j_fp)
            out = im.resize(new_size)
            out_fp = os.path.join(base_path, new_subdir, c, j)
            print(out_fp)
            out.save(out_fp)
```

executed in 8ms, finished 22:24:50 2019-12-11

In [209]:
```
def get_rgb_vectors(this_dir_fp):
    imlist = (io.imread_collection(this_dir_fp))
    res = np.zeros(shape=(1,3))
    for i in range(len(imlist)):
        m=transform.resize(imlist[i],(new_size[0],new_size[1],3))
        # re-shape to make list of RGB vectors.
        arr=m.reshape((new_size[0]*new_size[1]),3)
        # consolidate RGB vectors of all images
        res = np.concatenate((res,arr),axis=0)
    res = np.delete(res, (0), axis=0)
    return(res)
```

executed in 8ms, finished 09:42:28 2019-12-12

In [5]:
```
benign_dir_p = os.path.join(base_path, new_subdir, cat_ls[0]) + '/ISIC*.jpg'
benign_rgb_v = get_rgb_vectors(benign_dir_p)
print(len(benign_rgb_v))
malignant_dir_p = os.path.join(base_path, new_subdir, cat_ls[1]) + '/ISIC*.jpg'
malignant_rgb_v = get_rgb_vectors(malignant_dir_p)
print(len(malignant_rgb_v))
```

executed in 5m 27s, finished 22:30:22 2019-12-11

```
58373400
58373400
```

1.3.1.2 reshaping images into vectors [...]

(hiding to make report clean)

1.3.1.3 make training and test sets

In [12]:
```
train_mb, test_mb, train_lbl, test_lbl = train_test_split(mb_rgb_m, mb_lbl_m, test_size=0.2, random_state=0)
```

executed in 5.59s, finished 22:30:53 2019-12-11

1.3.1.4 normalize each channel [...]

(hiding to make report clean)

1.3.1.5 execute PCA

In [15]:
```
from sklearn.decomposition import PCA# Make an instance of the Model
pca_r = PCA(.975)
pca_g = PCA(.975)
pca_b = PCA(.975)
```

executed in 681ms, finished 22:31:51 2019-12-11

In [16]:
```
pca_r.fit(train_mbrs)
pca_g.fit(train_mbgs)
pca_b.fit(train_mbbs)
```

executed in 3m 8s, finished 22:35:02 2019-12-11

Out[16]:
```
PCA(copy=True, iterated_power='auto', n_components=0.975, random_state=None,
    svd_solver='auto', tol=0.0, whiten=False)
```

In [17]:
```
print(pca_r.n_components_)
print(pca_g.n_components_)
print(pca_b.n_components_)
```

executed in 448ms, finished 22:35:32 2019-12-11

```
65
95
107
```

In [18]:
```
train_mbrt = pca_r.transform(train_mbrs)
train_mbgt = pca_g.transform(train_mbgs)
train_mbbt = pca_b.transform(train_mbbs)
```

executed in 9.41s, finished 22:35:47 2019-12-11

In [19]:
```
test_mbrt = pca_r.transform(test_mbrs)
test_mbgt = pca_g.transform(test_mbgs)
test_mbbt = pca_b.transform(test_mbbs)
```

executed in 8.29s, finished 22:35:59 2019-12-11

1.3.1.6 combine the three channels (R, G, and B) back together [...]

(hiding to make report clean)

1.3.2 Initial evaluation of classification methods with pixel-based features

Now we have our training and test data sets and are ready for our models. I will try LDA, RandomForests and Support Vector Machine.

In [22]:
```
# create all the machine learning models
num_trees = 90
seed      = 3
models = []
models.append(('LDA', LinearDiscriminantAnalysis()))
models.append(('RF', RandomForestClassifier(n_estimators=num_trees, random_state=seed)))
models.append(('SVM', SVC(random_state=seed)))
```

executed in 772ms, finished 22:36:09 2019-12-11

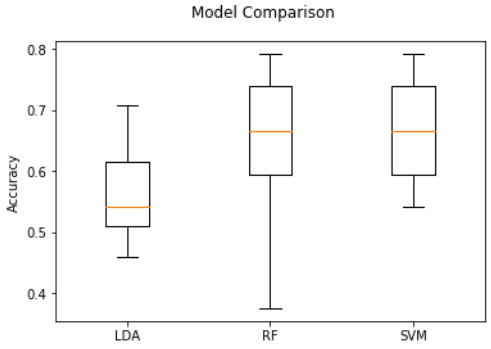
In [23]:
```
# 10-fold cross validation
results = []
names = []
for name, model in models:
    curr_kfold = KFold(n_splits=10)
    cv_results = cross_val_score(model, train_mbrgbt, train_lbl, cv=curr_kfold, scoring="accuracy")
    results.append(cv_results)
    names.append(name)
    message = "%s: mean = %f, std= %f" % (name, cv_results.mean(), cv_results.std())
    print(message)
```

executed in 7.49s, finished 22:36:20 2019-12-11

```
LDA: mean = 0.566667, std= 0.085797
RF: mean = 0.645833, std= 0.116741
SVM: mean = 0.662500, std= 0.084266
```

In [24]:
```
# boxplot model comparison
fig = plt.figure()
fig.suptitle('Model Comparison')
ax = fig.add_subplot(111)
plt.boxplot(results)
ax.set_xticklabels(names)
plt.ylabel("Accuracy")
plt.show()
```

executed in 4.92s, finished 22:36:29 2019-12-11



Now try models on test data

1.3.3 Fitting test data and tuning with pixel-based features

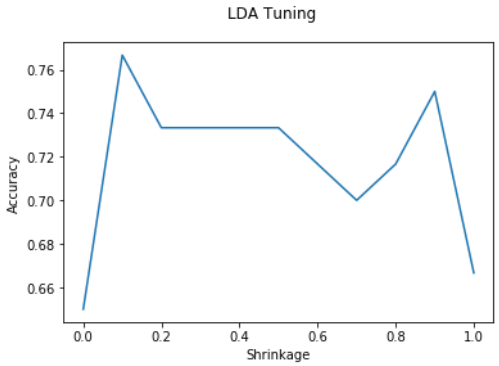
With LDA, I can tune the model by adjusting the shrinkage (as long as I employ the 'lsqr' or 'eigen'). I will use 'lsqr'. A shrinkage value of 0.1 seems to provide the best accuracy. This suggests that I may not have needed so many principal components.

In [424]:
```
lda_accuracy = []
shrinkage_ls = np.arange(0, 1.1, 0.1)
shrinkage_ls

for s in shrinkage_ls:
    my_lda = LinearDiscriminantAnalysis(solver='lsqr',shrinkage=s)
    my_lda.fit(train_mbrgbt, train_lbl)
    my_lda_predict = my_lda.predict(test_mbrgbt)
    my_lda_accuracy = accuracy_score(test_lbl, my_lda_predict, normalize=True)
    lda_accuracy.append(my_lda_accuracy)

fig = plt.figure()
fig.suptitle('LDA Tuning')
ax = fig.add_subplot(111)
plt.plot(shrinkage_ls, lda_accuracy)
plt.ylabel("Accuracy")
plt.xlabel("Shrinkage")
plt.show()
```

executed in 611ms, finished 14:32:22 2019-12-12



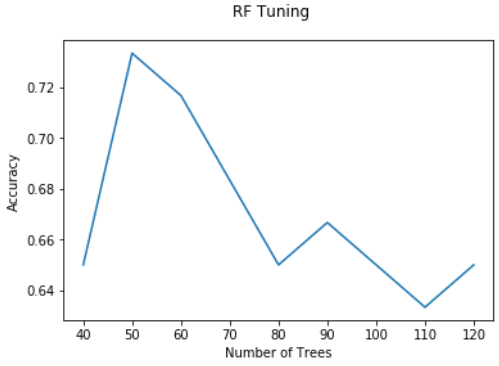
With RF, I can tune the model by adjusting the number of trees. The accuracy is best with 50 trees. This also suggests that I may not have needed so many principal components.

In [26]:
```
rf_accuracy = []
ntrees_ls = np.arange(40, 130, 10)
ntrees_ls

for nt in ntrees_ls:
    my_rf = RandomForestClassifier(n_estimators=nt, random_state=seed)
    my_rf.fit(train_mbrgbt, train_lbl)
    my_rf_predict = my_rf.predict(test_mbrgbt)
    my_rf_accuracy = accuracy_score(test_lbl, my_rf_predict, normalize=True)
    rf_accuracy.append(my_rf_accuracy)

fig = plt.figure()
fig.suptitle('RF Tuning')
ax = fig.add_subplot(111)
plt.plot(ntrees_ls, rf_accuracy)
plt.ylabel("Accuracy")
plt.xlabel("Number of Trees")
plt.show()
```

executed in 2.39s, finished 22:36:39 2019-12-11



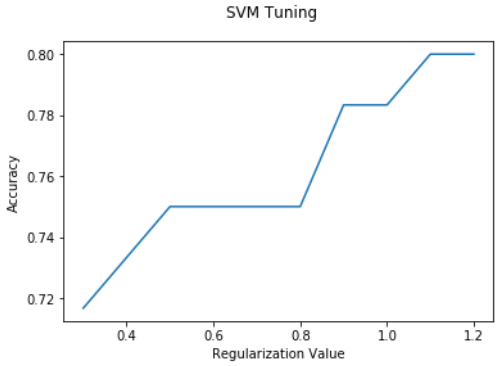
With SVM, I can tune the regularization value. With SVM, the accuracy improves with increasing regularization. Again, this suggests that I can try fewer principal components.

In [27]:
```
#kernel_ls = ['linear', 'poly', 'rbf', 'sigmoid']
reg_ls = np.arange(0.3, 1.3, 0.1)
svm_accuracy = []

for reg in reg_ls:
    my_svm = SVC(C=reg)
    my_svm.fit(train_mbrgbt, train_lbl)
    my_svm_predict = my_svm.predict(test_mbrgbt)
    my_svm_accuracy = accuracy_score(test_lbl, my_svm_predict, normalize=True)
    svm_accuracy.append(my_svm_accuracy)


fig = plt.figure()
fig.suptitle('SVM Tuning')
ax = fig.add_subplot(111)
plt.plot(reg_ls, svm_accuracy)
plt.ylabel("Accuracy")
plt.xlabel("Regularization Value")
plt.show()
```

executed in 479ms, finished 22:36:44 2019-12-11



1.3.4 Take-aways with pixel-based features

Of the three methods, the SVM method proved most accurate. For each method, adjusting the regularization allowed for tuning of the models. It appears that lowering the threshold for choosing principal components would have been and option. I could have used a lower percentage target than 97.5% to discrimate in selecting principal components and reducing the feature set.

1.4 Part 2 - Classification with Non-Pixel-Based Features

1.4.1 Literature Review

A quick internet search about how to distinguish benign and malignant skin features resulted in several potential indicators that may be employed in this exercise. A good resource is https://www.webmd.com/melanoma-skin-cancer/ss/skin-cancer-and-skin-lesions-overview (https://www.webmd.com/melanoma-skin-cancer/ss/skin-cancer-and-skin-lesions-overview) . The indicators I would like to use in the non-pixel based model would be the symmetry of the lesion and the color variation of the lesion in the different channels.

1.4.2 Feature Engineering

Symmetry

To quantify symmetry, I first use the blob functions from skimage to detect the lesions if possible. I take the blob closest to the center and crop with that as a center. I try to avoid cropping too much. But cropping can help to throw out non-useful dark borders which some images have. I cut the cropped images in half - once vertically and once horizontally. Then I use the structural similarity score to see how well they match. The higher the scores, I assume the greater the symmetry. Of course it's possible for a lesion to be symmetric and oblong and positioned diagonally and so my method may benefit from image rotation but I ran out of time to do so.

Color Variation

I also converted each image into gray scale, red channel, green channel and blue channel. For each of those, I calculate the mean and standard deviation of the itensities. I also take each of those and apply kMeans clustering (with n=2) to extract the background and lesion groups (light and dark regions) to also get the mean and standard deviations. So to capture color variation I have 24 features: the mean and standard deviation for each type (greyscale and RGB) as well as the mean and standard deviation for each of the dark and light regions of those four types.

Final Feature Set

Finally, I combine the two symmetry features and the 24 color variation features into a large matrix. I normalize each feature and then can use them with the same sort of classifiers I employed in the pixel-based effort (LDA, RF, and SVM).

In [28]:
```
def distance(p0, p1):
    return(math.sqrt((p0[0] - p1[0])**2 + (p0[1] - p1[1])**2))
```
executed in 98ms, finished 22:36:47 2019-12-11

In [333]:
```
def get_best_center(blob_set, img_size):
    img_center_loc = (img_size[0]/2, img_size[1]/2)
    best_dist = distance(img_size, img_center_loc)
    best_center = img_size
    for b in blob_set:
        y, x, r = b
        blob_center_loc = (x, y)
        blob_center_dist = distance(blob_center_loc, img_center_loc)
        if blob_center_dist < best_dist:
            best_dist = blob_center_dist
            best_center = blob_center_loc
    max_dist = distance(img_size, img_center_loc) / 4
    if (best_dist >= max_dist):
        best_center = img_center_loc
    if (best_center[0] == 0):
        best_center = img_center_loc
    if (best_center[1] == 0):
        best_center = img_center_loc
    return(best_center)
```
executed in 10ms, finished 11:31:08 2019-12-12

In [334]:
```
def best_center_crop(img_to_crop, new_center, img_size):
    #print(img_size)
    #print(new_center)
    w1 = new_center[0] - 0
    w2 = img_size[0]-new_center[0]
    #print(w1, w2)
    best_w = min(w1, w2)
    #print(best_w)
    h1 = new_center[1] - 0
    h2 = img_size[1]-new_center[0]
    #print(h1, h2)
    best_h = min(h1, h2)
    #print(best_h)
    w_start = np.int(new_center[0] - best_w)
    w_nd = np.int(new_center[0] + best_w)
    h_start = np.int(new_center[1] - best_h)
    h_nd = np.int(new_center[1] + best_h)
    #print(w_start,w_nd,h_start,h_nd)
    return(img_to_crop[h_start:h_nd,w_start:w_nd])


t = best_center_crop(imlist[0], (353.0, 269.0), new_size)
print(t.shape)
```
executed in 19ms, finished 11:31:10 2019-12-12

```
(378, 706, 3)
```

In [369]:
```
def cut_image(img_to_cut, direction):
    height = img_to_cut.shape[0]
    width = img_to_cut.shape[1]
    #print(height, width)
    if (direction=='vertical'):
        h_nd = np.int(height / 2.0)
        top_img = img_to_cut[0:h_nd,:]
        bottom_img = img_to_cut[h_nd:,:]
        flip_bottom_img = bottom_img[::-1,:]
        top_height = top_img.shape[0]
        #ntop_img = transform.resize(top_img,(top_height,width,3))
        #nflip_bottom_img=transform.resize(flip_bottom_img,(top_height,width,3))
        nflip_bottom_img = flip_bottom_img[0:h_nd,:]
        return([top_img, nflip_bottom_img])
    elif (direction=='horizontal'):
        w_nd = np.int(width / 2.0)
        left_img = img_to_cut[:,0:w_nd]
        right_img = img_to_cut[:,w_nd:]
        flip_right_img = right_img[:,::-1]
        left_width = left_img.shape[1]
        #nleft_img =transform.resize(left_img,(height,left_width,3))
        #nflip_right_img =transform.resize(flip_right_img,(height,left_width,3))
        nflip_right_img = flip_right_img[:,0:w_nd]
        return([left_img,nflip_right_img])
```
executed in 8ms, finished 12:17:31 2019-12-12

In [377]:
```
def get_channel_variations(c_vec):
    c_vec_r = c_vec.reshape(-1,1)
    v_mean = np.mean(c_vec_r)
    v_std = np.std(c_vec_r)
    km_results = KMeans(n_clusters = 2, random_state=0).fit(c_vec_r)
    big_label = np.argmax(km_results.cluster_centers_)
    big_idx = np.where(km_results.labels_ == big_label)[0]
    small_idx = np.where(km_results.labels_ != big_label)[0]
    big_clstr_mean = np.mean(c_vec_r[big_idx])
    big_clstr_std = np.std(c_vec_r[big_idx])
    small_clstr_mean = np.mean(c_vec_r[small_idx])
    small_clstr_std = np.std(c_vec_r[small_idx])
    return([v_mean, v_std, big_clstr_mean, big_clstr_std, small_clstr_mean, small_clstr_std])
```

executed in 7ms, finished 12:30:53 2019-12-12

In [403]:
```
def get_np_features(this_dir_fp):
    imlist = (io.imread_collection(this_dir_fp))
    print(len(imlist))
    res = np.zeros(26)
    for i in range(len(imlist)):
        print(i)
        this_img = imlist[i]
        inv_img = invert(this_img, signed_float=False)
        image_gray = rgb2gray(inv_img)
        blobs_dog = blob_dog(image_gray, max_sigma=350, min_sigma=50, threshold=.2)
        img_best_center = get_best_center(blobs_dog, new_size)
        #print(img_best_center)
        cropped_img = best_center_crop(this_img, img_best_center, new_size)
        cropped_img_vert = cut_image(cropped_img, "vertical")
        cropped_img_hor = cut_image(cropped_img, "horizontal")
        vert_sym =  structural_similarity(cropped_img_vert[0], cropped_img_vert[1], multichannel=True)
        hor_sym =  structural_similarity(cropped_img_hor[0], cropped_img_hor[1], multichannel=True)
        rchv = cropped_img[:,:,0][0]
        r_vars = get_channel_variations(rchv)
        gchv = cropped_img[:,:,1][0]
        g_vars = get_channel_variations(gchv)
        bchv = cropped_img[:,:,2][0]
        b_vars = get_channel_variations(bchv)
        grchv = 0.2125 * rchv  + 0.7154 * gchv + 0.0721 * bchv
        gr_vars = get_channel_variations(grchv)
        out_vars = np.concatenate(([vert_sym, hor_sym], r_vars), axis=None)
        out_vars = np.concatenate((out_vars, g_vars), axis=None)
        out_vars = np.concatenate((out_vars, b_vars), axis=None)
        out_vars = np.concatenate((out_vars, gr_vars), axis=None)
        res = np.concatenate((res,out_vars),axis=0)
        if (i % 25 == 0):
            print(out_vars)
    #res = np.delete(res, (0), axis=0)
    return(res)
```

executed in 14ms, finished 13:44:54 2019-12-12

1.4.2.1 use above functions to get features for each image in benign and malignant set [...]

(hiding to make report clean)

1.4.2.2 make train and test data sets with features

In [408]:
```
train_mbnp, test_mbnp, train_lbl, test_lbl = train_test_split(mb_npf, mb_lbl_m, test_size=0.2, random_state=0)
```

executed in 5ms, finished 14:12:38 2019-12-12

In [409]:
```
scaler_np = StandardScaler() # Fit on training set only.
scaler_np.fit(train_mbnp)    # Apply transform to both the training set and the test set.

train_mbnps = scaler_np.transform(train_mbnp)
test_mbnps = scaler_np.transform(test_mbnp)
```

executed in 43ms, finished 14:12:49 2019-12-12

1.4.3 Initial evaluation of classification methods with non-pixel-based features

Now we have our training and test data sets and are ready for our models. I will again try LDA, RandomForests and Support Vector Machine.

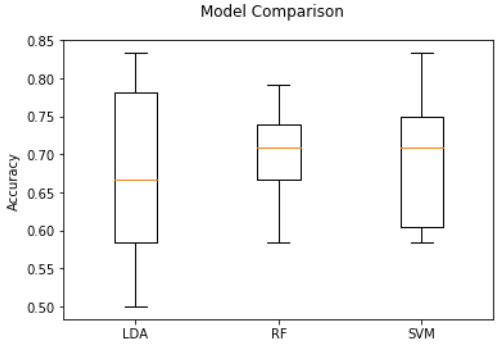
In [410]:
```
# 10-fold cross validation
results = []
names = []
for name, model in models:
    curr_kfold = KFold(n_splits=10)
    cv_results = cross_val_score(model, train_mbnps, train_lbl, cv=curr_kfold, scoring="accuracy")
    results.append(cv_results)
    names.append(name)
    message = "%s: mean = %f, std= %f" % (name, cv_results.mean(), cv_results.std())
    print(message)
```

executed in 2.44s, finished 14:13:11 2019-12-12

```
LDA: mean = 0.670833, std= 0.112500
RF: mean = 0.695833, std= 0.059073
SVM: mean = 0.695833, std= 0.087500
```

In [411]:
```
# boxplot model comparison
fig = plt.figure()
fig.suptitle('Model Comparison')
ax = fig.add_subplot(111)
plt.boxplot(results)
ax.set_xticklabels(names)
plt.ylabel("Accuracy")
plt.show()
```

executed in 317ms, finished 14:13:15 2019-12-12



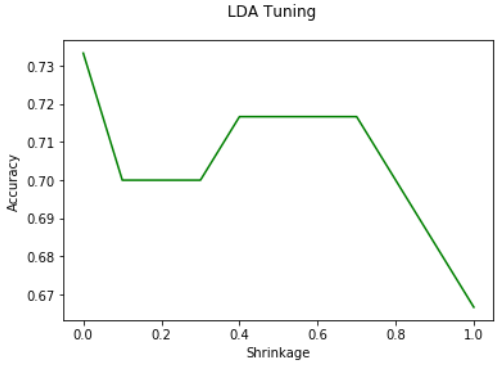
1.4.4 Fitting test data and tuning with non-pixel-based features

In [415]:
```
lda_accuracy = []
shrinkage_ls = np.arange(0, 1.1, 0.1)
shrinkage_ls

for s in shrinkage_ls:
    my_lda = LinearDiscriminantAnalysis(solver='lsqr',shrinkage=s)
    my_lda.fit(train_mbnps, train_lbl)
    my_lda_predict = my_lda.predict(test_mbnps)
    my_lda_accuracy = accuracy_score(test_lbl, my_lda_predict, normalize=True)
    lda_accuracy.append(my_lda_accuracy)

fig = plt.figure()
fig.suptitle('LDA Tuning')
ax = fig.add_subplot(111)
plt.plot(shrinkage_ls, lda_accuracy, color='green')
plt.ylabel("Accuracy")
plt.xlabel("Shrinkage")
plt.show()
```

executed in 274ms, finished 14:18:30 2019-12-12

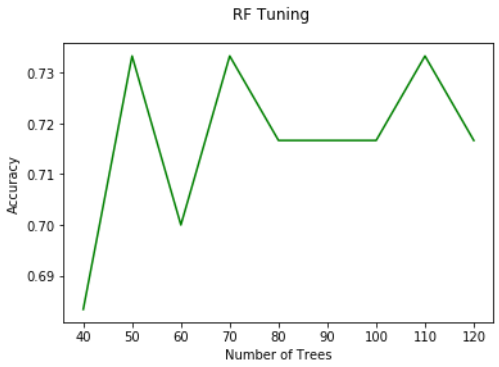


In [413]:
```
rf_accuracy = []
ntrees_ls = np.arange(40, 130, 10)
ntrees_ls

for nt in ntrees_ls:
    my_rf = RandomForestClassifier(n_estimators=nt, random_state=seed)
    my_rf.fit(train_mbnps, train_lbl)
    my_rf_predict = my_rf.predict(test_mbnps)
    my_rf_accuracy = accuracy_score(test_lbl, my_rf_predict, normalize=True)
    rf_accuracy.append(my_rf_accuracy)

fig = plt.figure()
fig.suptitle('RF Tuning')
ax = fig.add_subplot(111)
plt.plot(ntrees_ls, rf_accuracy, color="green")
plt.ylabel("Accuracy")
plt.xlabel("Number of Trees")
plt.show()
```

executed in 1.76s, finished 14:13:26 2019-12-12

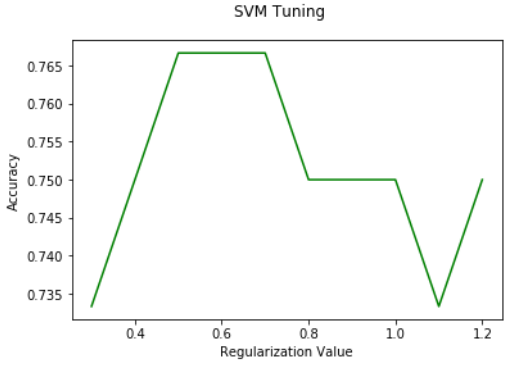


In [414]:
```
#kernel_ls = ['linear', 'poly', 'rbf', 'sigmoid']
reg_ls = np.arange(0.3, 1.3, 0.1)
svm_accuracy = []

for reg in reg_ls:
    my_svm = SVC(C=reg)
    my_svm.fit(train_mbnps, train_lbl)
    my_svm_predict = my_svm.predict(test_mbnps)
    my_svm_accuracy = accuracy_score(test_lbl, my_svm_predict, normalize=True)
    svm_accuracy.append(my_svm_accuracy)


fig = plt.figure()
fig.suptitle('SVM Tuning')
ax = fig.add_subplot(111)
plt.plot(reg_ls, svm_accuracy, color="green")
plt.ylabel("Accuracy")
plt.xlabel("Regularization Value")
plt.show()
```

executed in 286ms, finished 14:13:29 2019-12-12



1.4.5 Take-aways with non-pixel-based features

The three classification methods seem to provide more similar results to one another than with the pixel-based features. Again SVM provided the best accuracy but not by much compared to the pixel-based effort.