COMP 432 Assignment 1

In this assignment you'll be clustering and classifying emojis!

The assignment is very much like a lab. It is broken into many small and specific steps that build on each other. The main differences are:

1. the assignment combines multiple concepts together,
2. the assignment will be carefully graded, and
3. where **specified**, you must add comments to your code.

There are 5 questions: Q1 (20 marks), Q2 (20 marks), Q3 (20 marks), Q4 (10 marks), Q5 (10 marks) and one bonus question Q6 (5 bonus marks).

Rules for academic integrity:

- Like labs, students are encouraged to ask conceptual questions of TAs and of other students, and can answer each others' conceptual questions.
- Unlike labs, students are not allowed to post example code in a public forum, even if the code is wrong; code and pseudocode can only be shared with TAs when requesting help.
- Never ask for, or offer, code snippets for the assignment to your fellow students. Doing so is forbidden, and is a major violation of academic integrity, both of the person who shared the code and the person who accepted the code. Violations of academic integrity will be reported to the Dean's Office. Violators risk their academic standing.

Advice:

- *Invest in plotting.* Plotting is super important for ML and for data sciences generally. So, put in the time to learn how to make good plots, efficiently!
- *Always set random seeds.* Some steps of the assignment involve randomness, even if you do not explicitly ask for it. In order to make your assignment reproducible, you must set scikit-learn's *random_state* to a constant (e.g., to 0) whenever applicable.
- *Save your notebook frequently.* Although Jupyter notebooks are mostly reliable, it is possible to encounter an erroneous state, where the most recent changes cannot be saved to disk by the notebook's own save functionality.

Run the code cell below to import the necessary packages.

In [1]:
```
import os                           # for os.path.exists
import json                         # for loading metadata
import urllib                       # for downloading remote files
import numpy as np
import matplotlib as mpl
import matplotlib.pyplot as plt
import sklearn
import sklearn.tree
import sklearn.metrics
import sklearn.ensemble
import sklearn.preprocessing

# Matplotlib might complain that a lot of figures are open, but suppress that war
plt.rcParams.update({'figure.max_open_warning': 0})

def roundVal(val):
    rounded = round(val,3)
    return rounded
```

Run the code cell below to define some utility functions for fetching data and for processing images.

In [27]:
```
def download(remoteurl: str, localfile: str):
    """
    Download remoteurl to localfile, unless localfile already exists.
    Returns the localfile string.
    """
    if not os.path.exists(localfile):
        print("Downloading %s..." % localfile)
        filename, headers = urllib.request.urlretrieve(remoteurl, localfile)
    return localfile

def rgba_to_rgb(image):
    """
    Converts image from RGBA format (H,W,4) to RGB format (H,W,3).
    Returns the new RGB image.
    """
    assert image.ndim == 3, "Expected 3-dimensional array"
    assert image.shape[2] == 4, "Expected 4 colour channels"
    assert image.max() <= 1, "Expected color values in range [0,1]"
    rgb, a = np.split(image, [3], axis=2)  # Split into (H,W,3) and (H,W,1)
    return a*rgb + (1-a)                   # Apply alpha blending to get RGB

def findEmojiName(x,y):
    for i in range(0, len(data)):
        if data[i]['sheet_x'] == x and data[i]['sheet_y'] == y:
            return data[i]['short_name']
```

Q1 — Download emoji metadata and images [20 marks total]

The image data and corresponding metadata that you need for this assignment is available from github.com/imcal/emoji-data (https://github.com/iamcal/emoji-data), where you can also find a description of the data. The specific files you'll need are only:

- *emoji.json*
- *sheets-clean/sheet_{vendor}_{size}_clean.png*

where *{vendor}* is one of *{apple, facebook, google, twitter}* and *{size}* is the pixel resolution. You'll need emojis from all four vendors, but only the small *16x16* pixel versions (to make training faster). However, do NOT download the files manually.

Q1a — Write code to download the files [5 marks]

Use the *download* function defined above to fetch the five files *procedurally*.

Hint: When you visit a Github URL in your browser, Github normally returns an HTML file for rendering in your web browser. To ask Github for an actual raw file (instead of the web page for displaying that file) you must use special URLs. If you view a file in your web browser https://github.com/iamcal/emoji-data/{path_to_file} (https://github.com/iamcal/emoji-data/%7Bpath_to_file%7D) then you should use URL https://github.com/iamcal/emoji-data/raw/master/{path_to_file} (https://github.com/iamcal/emoji-data/raw/master/%7Bpath_to_file%7D).

In [3]:
```
# Your code here. Use as many lines as you need.
# Feel free to define global variables like EMOJI_SIZE=16 for later use.
resources_dir = "resources"
EMOJI_SIZE = 16
vendors = ['apple', 'facebook', 'google', 'twitter']
size = 16
imageFileNames = []
if not os.path.exists(resources_dir):
    os.mkdir(resources_dir)
download("https://github.com/iamcal/emoji-data/raw/master/emoji.json" , "resource

for vendor in vendors:
    toDownload = 'sheet_' + vendor + '_' + str(size) + '_clean.png'
    imageFileNames.append("resources/"+toDownload)
    download(("https://github.com/iamcal/emoji-data/raw/master/sheets-clean/" + t
```

```
Downloading resources/emojis.json...
Downloading resources/sheet_apple_16_clean.png...
Downloading resources/sheet_facebook_16_clean.png...
Downloading resources/sheet_google_16_clean.png...
Downloading resources/sheet_twitter_16_clean.png...
```

Q1b — Load and inspect the emoji metadata [5 marks]

The emoji metadata is contained in a JSON file, which Python's **json (https://docs.python.org/3/library/json.html)** module can easily load and parse for you.

Write code to load the *emoji.json* file, then display the metadata for the first emoji (index 0) so that you can see an example. It should have short name '*hash*'.

In [20]:
```
# Your answer here. Aim for 2-4 lines.
# Keep the metadata in a global variable that you can keep referring to.
f = open('resources/emojis.json',)
data = json.load(f)
f.close()
print(data[0]["short_name"])
```

```
hash
```

Write code to find the index of the emoji having short name *'laughing'*, then display its metadata (the *dict* object). Do not use the *sort_order* field of the emoji metadata, it is not relevant to this assignment.

In [5]:
```
# Your answer here. Aim for 1-5 lines. Keep the index in a global variable for la
laughing_index = 0
for i in range(0, len(data)):
    if data[i]['short_name'] == 'laughing':
        laughing_index = (data[i]['sheet_x'] ,data[i]['sheet_y'])
        print(data[i])
```

```
{'name': 'SMILING FACE WITH OPEN MOUTH AND TIGHTLY-CLOSED EYES', 'unified': '1F
606', 'non_qualified': None, 'docomo': 'E72A', 'au': 'EAC5', 'softbank': None,
'google': 'FE332', 'image': '1f606.png', 'sheet_x': 32, 'sheet_y': 58, 'short_n
ame': 'laughing', 'short_names': ['laughing', 'satisfied'], 'text': None, 'text
s': [':>', ':->'], 'category': 'Smileys & Emotion', 'subcategory': 'face-smilin
g', 'sort_order': 5, 'added_in': '0.6', 'has_img_apple': True, 'has_img_googl
e': True, 'has_img_twitter': True, 'has_img_facebook': True}
```

Q1c — Load and inspect the emoji image data [10 marks]

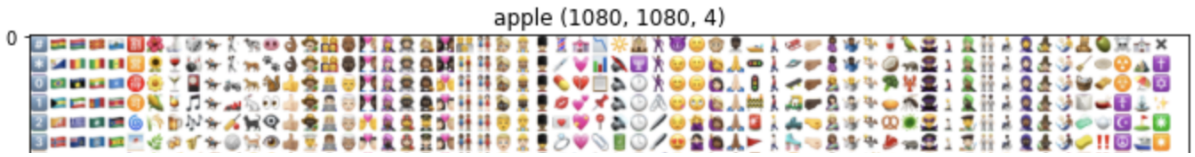
The emoji image data is contained in PNG files, which Matplotlib's **imread (https://matplotlib.org/3.3.2/api/_as_gen/matplotlib.pyplot.imread.html)** function can load as a Numpy array. The image format is RGBA (*red*, *green*, *blue*, *alpha*) where *alpha* determines the opacity of each pixel.

Write code to load the four emoji sheet images. The list of images should be in order *{apple,facebook,google,twitter}*.

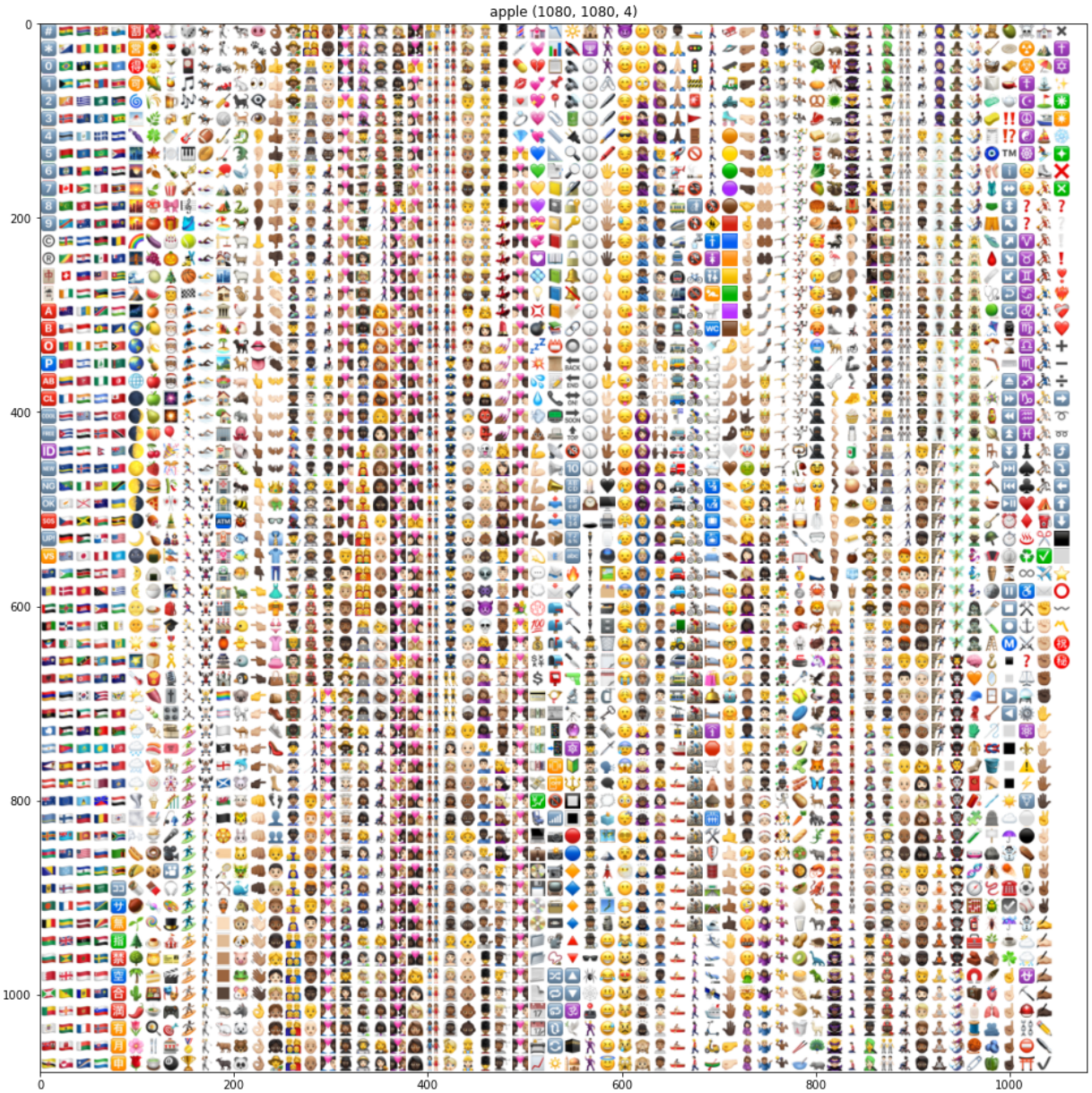
In [6]:
```
# Your answer here. Aim for 1-4 lines. You can define a global variable to hold t
emojiImageData = [plt.imread(fname) for fname in imageFileNames]
```

Write code to plot each vendor's sheet image. Generate four separate plots, where the title of each plot should be "*vendor* (*height*, *width*, *channels*)" where *height* and *width* are the size of the

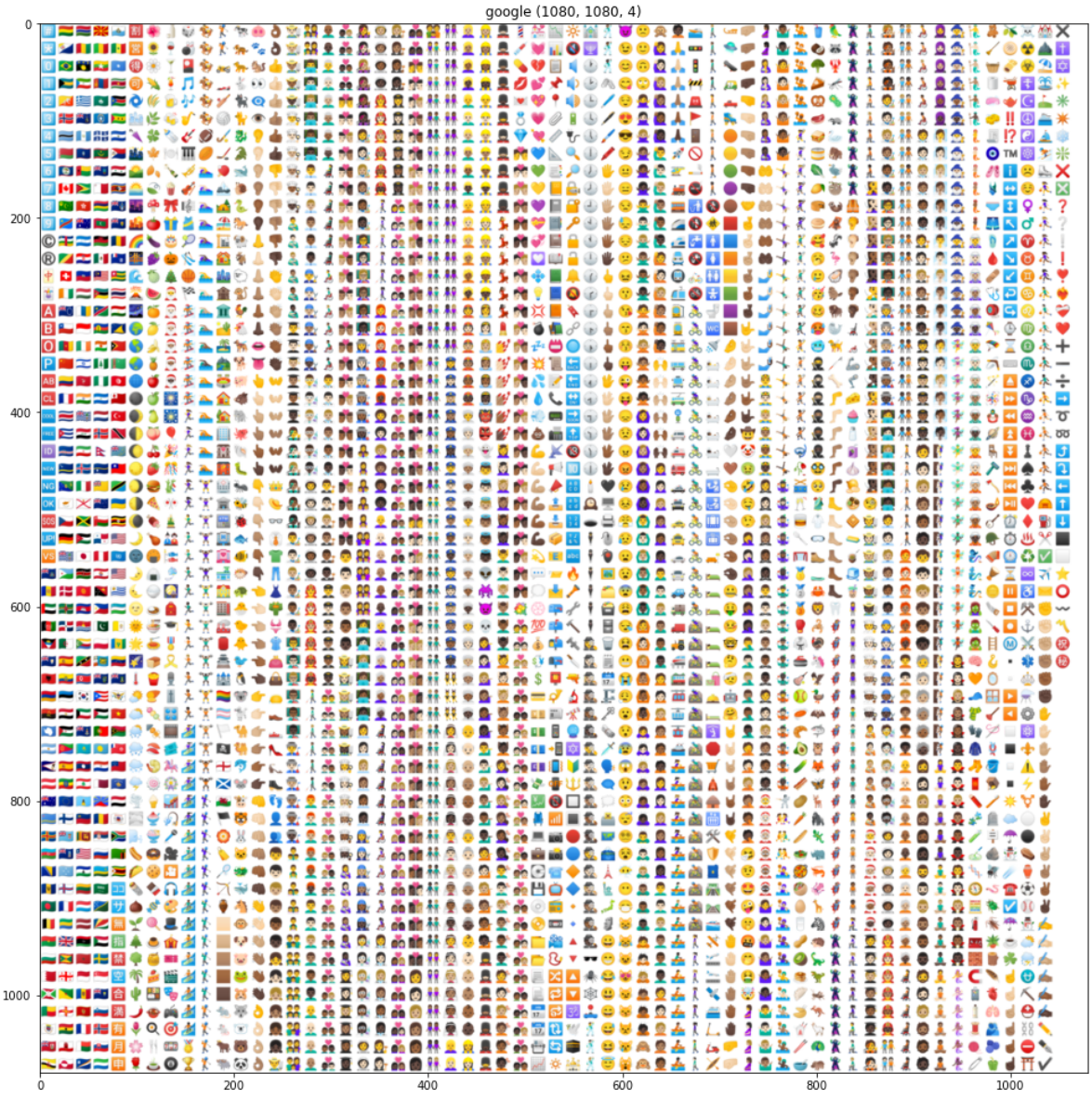
sheet and *channels* is the number of colour channels. Use the *figsize* argument of Matplotlib's *figure* function to enlarge the figures. The top of your first plot should look like this:

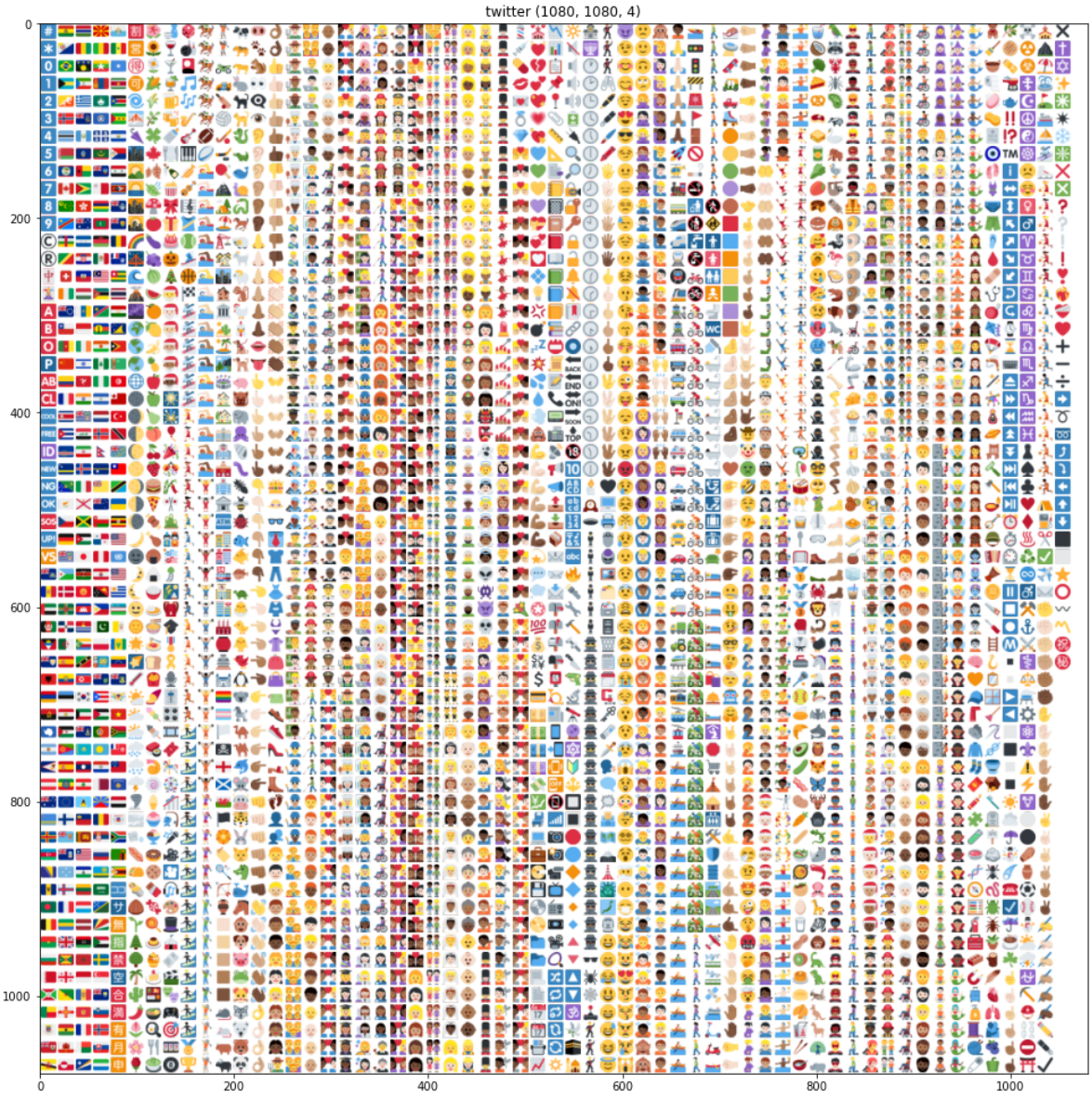


In [7]:
```
# Your answer here. Aim for 4-6 lines.
counter = 0
for vendor in vendors:
    plt.figure(figsize=(17,17))
    plt.imshow(emojiImageData[counter]);
    plt.title(vendor + ' ' + str(emojiImageData[counter].shape));
    counter += 1
```

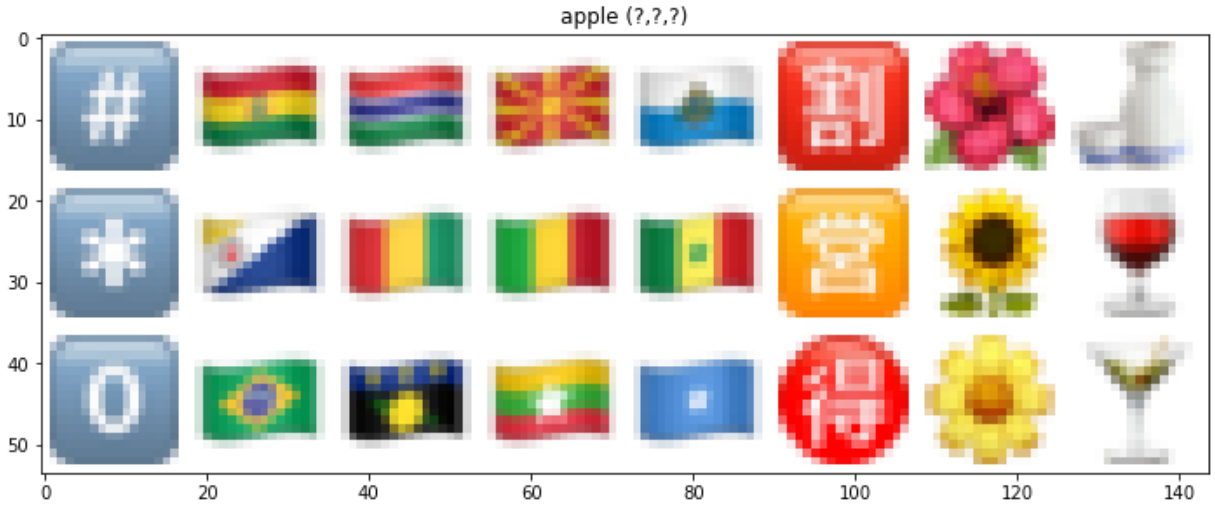



facebook (1080, 1080, 4)
0
200
400
600
800
1000
0
200
400
600
800
1000


google (1080, 1080, 4)
0
200
400
600
800
1000
0
200
400
600
800
1000

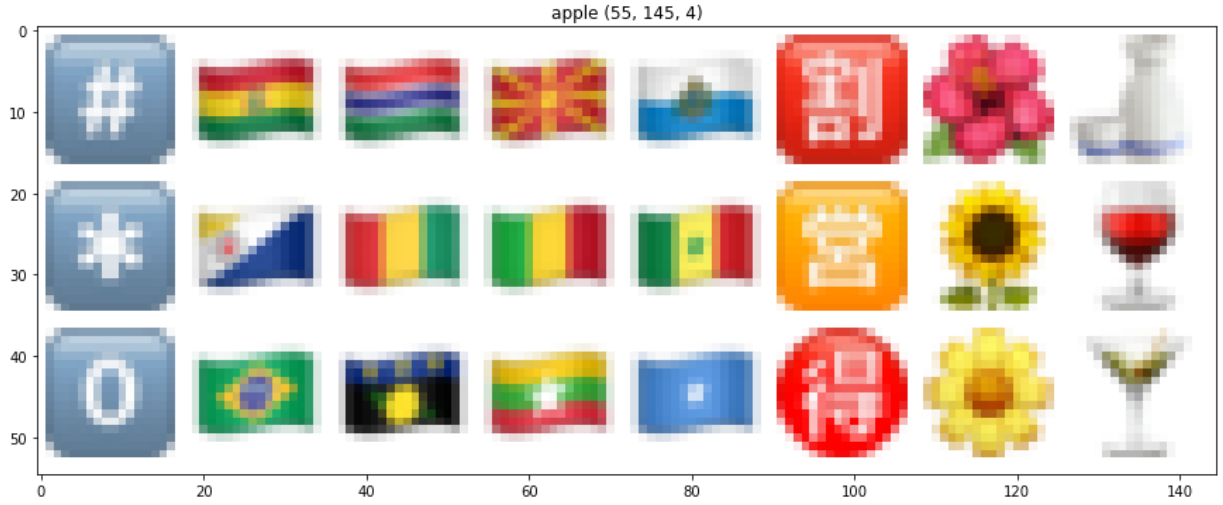


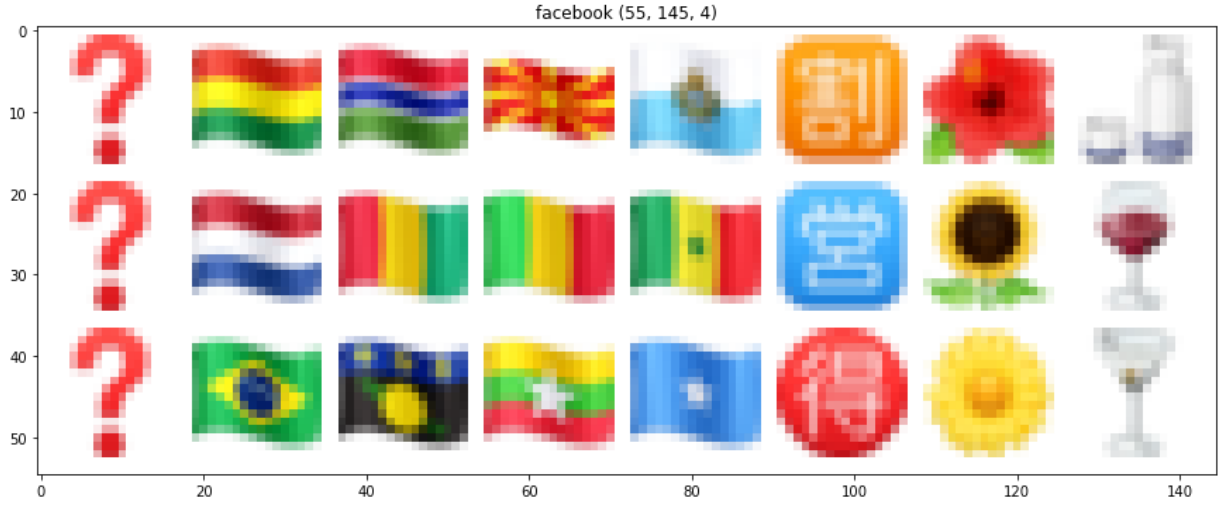
Write code to generate the same four plots as above, except use Numpy slicing to display only the first 3 rows and 8 columns of the sheet. To ensure you do not crop any emojis, take note of any "padding" between the 16x16 emojis in the sheet. Your first plot should look like this, but with the shape numbers (?) and axis ticks filled in:

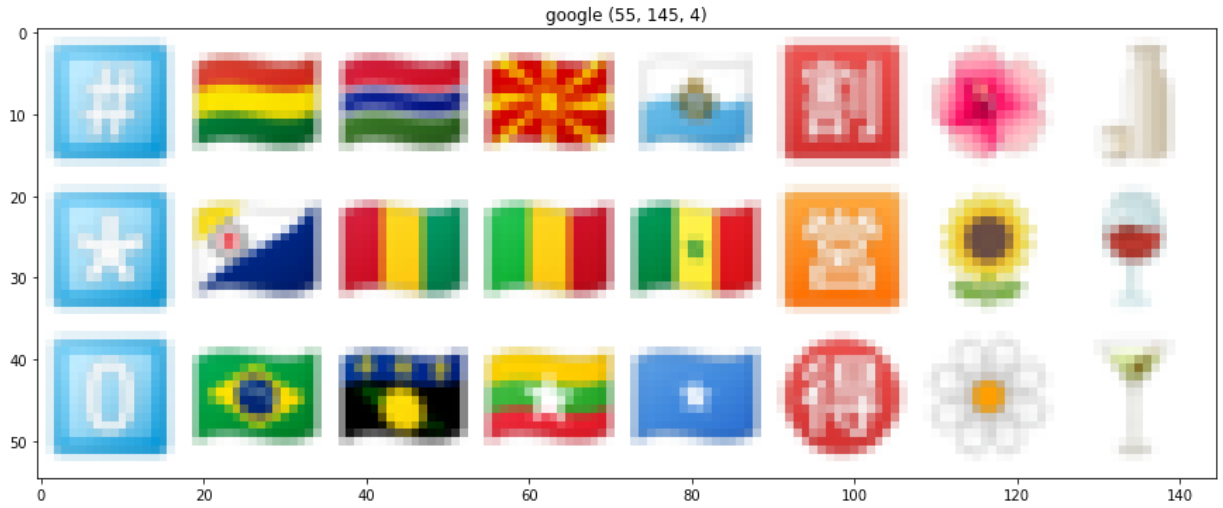


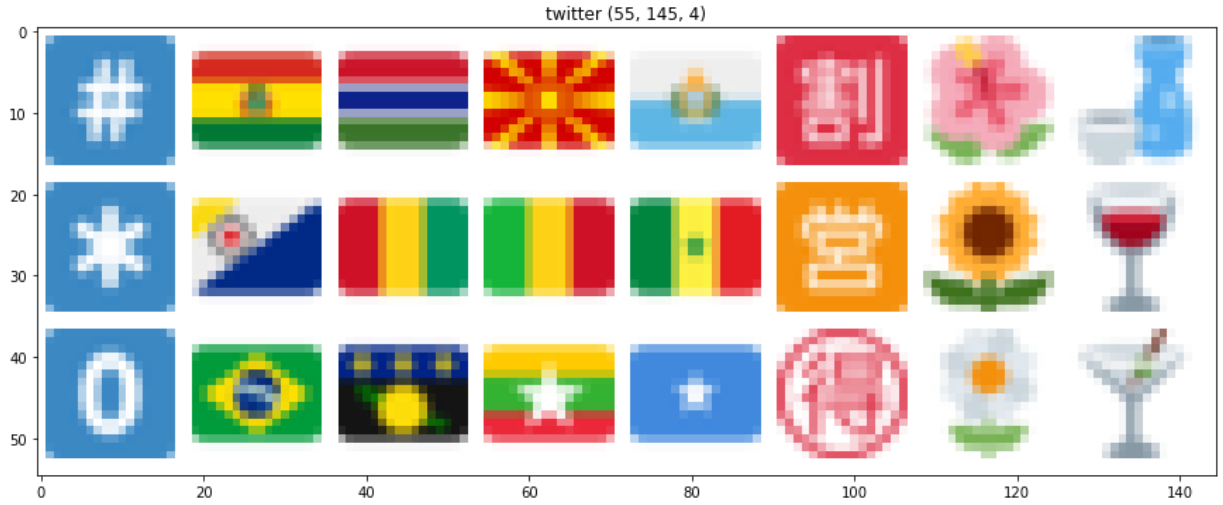
(If you see a red question mark like ? for a vendor, it means they do not provide that particular emoji.)

In [8]:
```
# Your answer here. Aim for 5-9 lines.
counter = 0
for vendor in vendors:
    toDisplay = emojiImageData[counter][:55,:145]
    plt.figure(figsize=(15,15))
    plt.imshow(toDisplay);
    plt.title(vendor + ' ' + str(toDisplay.shape));
    counter += 1
```





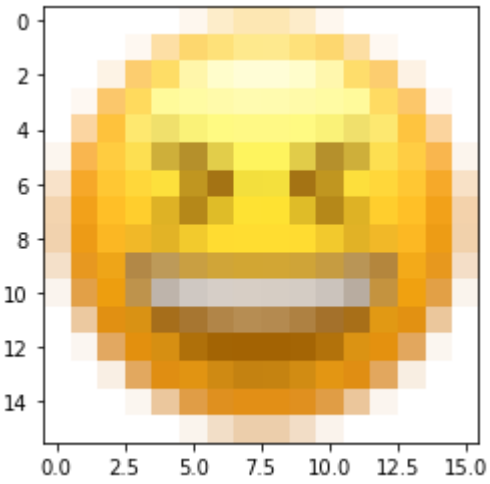




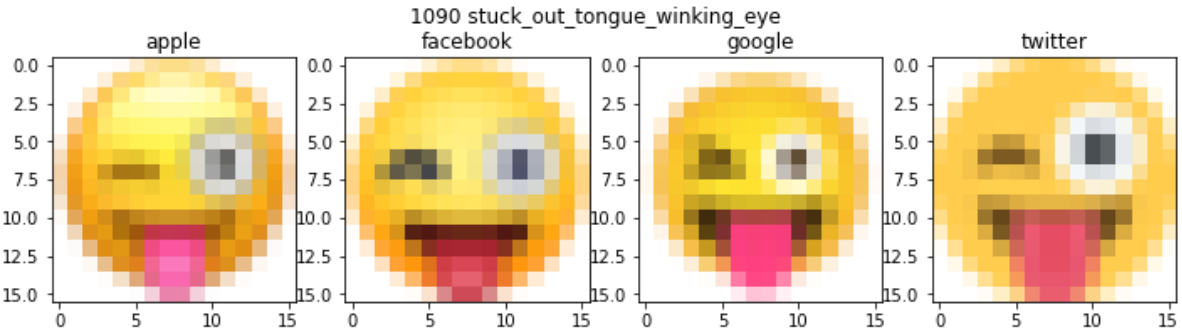
Implement *get_emoji_image*. It should extract a 16x16 RGBA emoji image by its style index (0=*apple*, 1=*facebook*, 2=*google*, 3=*twitter*) and emoji index (as they appear in *emoji.json*). Internally, your function may refer to any global variables you have already defined (metadata, images, size, padding). Use the *sheet_x* and *sheet_y* fields of the metadata. Use slicing and avoid for-loops. **Briefly comment each line of your code.**

In [185]:
```
def get_emoji_image(style_index, emoji_index):
    """
    Given a vendor style index (apple=0,facebook=1,google=2,twitter=3)
    and an emoji index, returns the 16x16 RGBA image as a Numpy array
    with shape (16,16,4).
    """
    # Your implementation here. Aim for 5-8 lines (not including comments).
    # the index x and y to start extracting emoji is at (1,1), so we have to incl
    # each emoji has a space of 2 units, so we have to do 2 x emoji_index in orde
    x_init = 1+16*emoji_index[0] + 2*emoji_index[0]
    x_end = x_init + 16
    # the length of an emoji is 16, so mark the end of emoji position at x_init+1
    y_init = 1+16*emoji_index[1] + 2*emoji_index[1]
    y_end = y_init + 16
    img_ar = emojiImageData[style_index][y_init:y_end,x_init:x_end]
    print(img_ar.shape)
    return img_ar

plt.imshow(get_emoji_image(0,laughing_index));
```

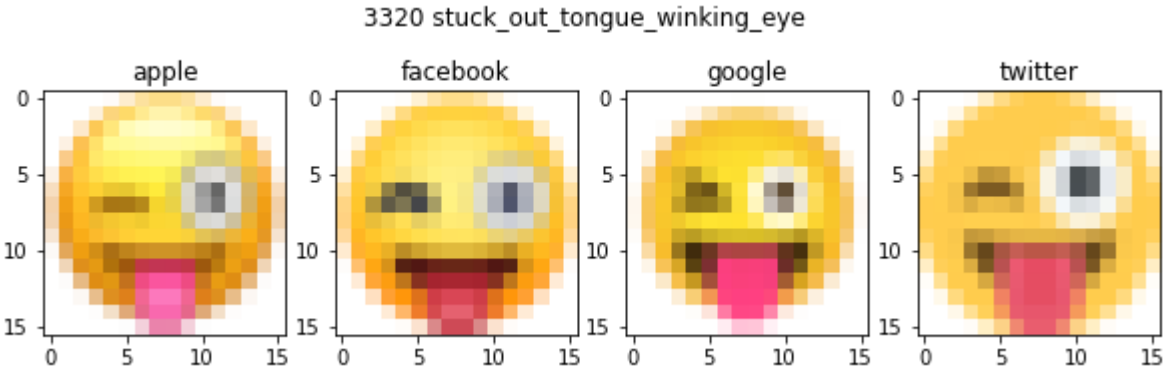
```
(16, 16, 4)
```



Implement *plot_emoji_styles*. Implement the *plot_emoji_styles* function below, using your *get_emoji_image* function as a subroutine. Use *figsize* to control the size of your figure, use Matplotlib's *subplot* and *title* functions along with its **suptitle (https://matplotlib.org/3.3.2/api/_as_gen/matplotlib.pyplot.suptitle.html)** to create titles that show the emoji index, the emoji short name, and the vendor title above each style, as shown below:

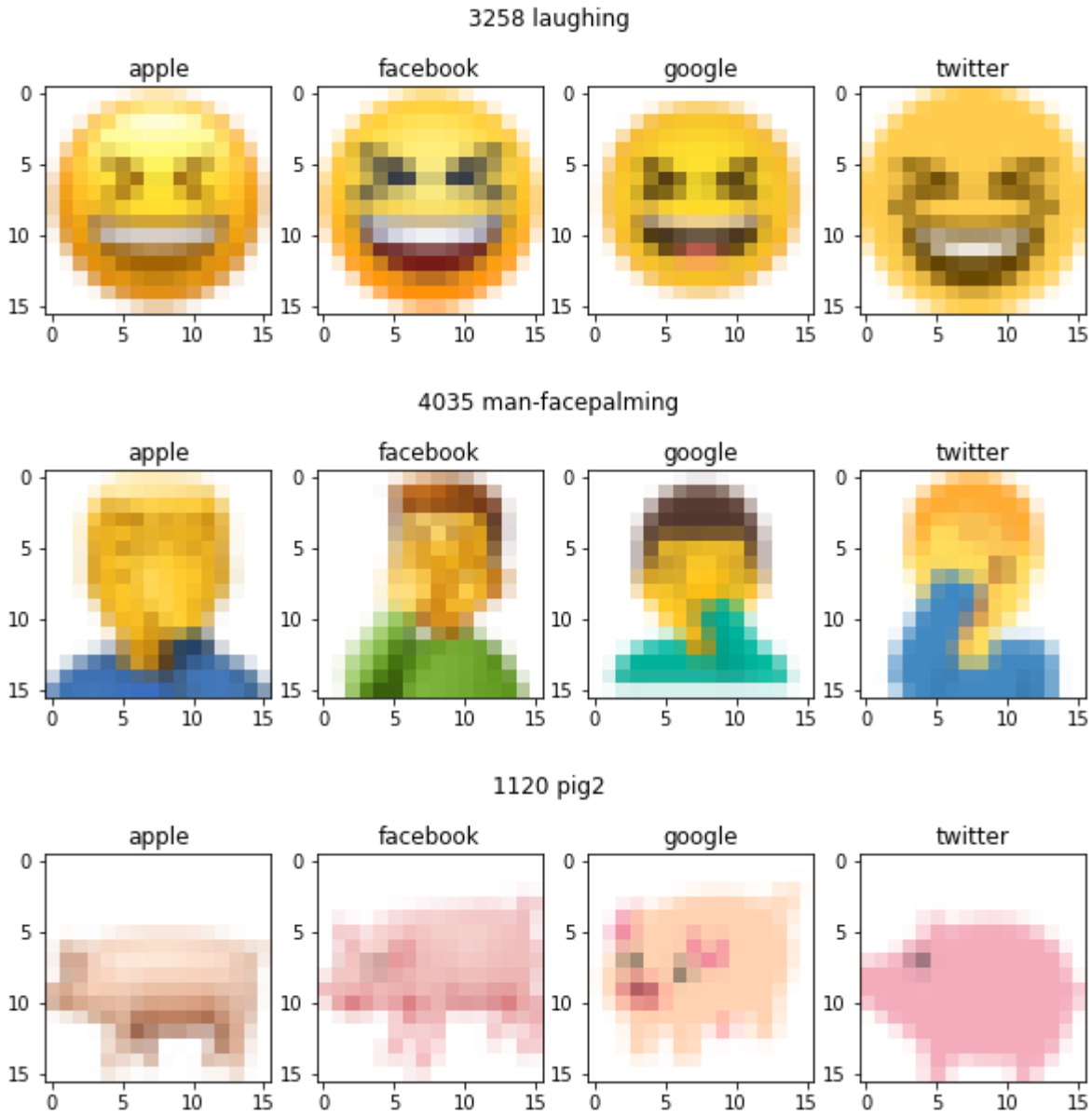


In [29]:
```
def plot_emoji_styles(emoji_index):
    """Plots all four vendor styles of the given emoji."""
    # Your implementation here. Aim for 6-8 lines.
    plt.figure(figsize = (10,3))
    plt.suptitle((str(emoji_index[0]) + '' + str(emoji_index[1])) + ' ' + findEmo
    for i in range(0,4):
        plt.subplot(1,4,i+1)
        plt.imshow(get_emoji_image(i, emoji_index));
        plt.title(vendors[i])
    plt.show()

plot_emoji_styles((33,20))
```



Run your *plot_emoji_styles* function to plot the '*laughing*' emoji from **Q1b**. Also plot two other emojis of your choosing. (Except poop. You're not allowed to plot the poop emoji. Don't you dare. No, no wait stop, have some self-respect, don't do it, noooo!)

In [30]:
```
# Your code here. Aim for 2-3 lines.
plot_emoji_styles(laughing_index)
plot_emoji_styles((40,35))
plot_emoji_styles((11,20))
```



Q2 — Build an emoji dataset for machine learning [20 marks total]

This question is about converting your list of four raw image sheets into a dataset $(\boldsymbol{X}, \boldsymbol{y})$ suitable for training with scikit-learn.

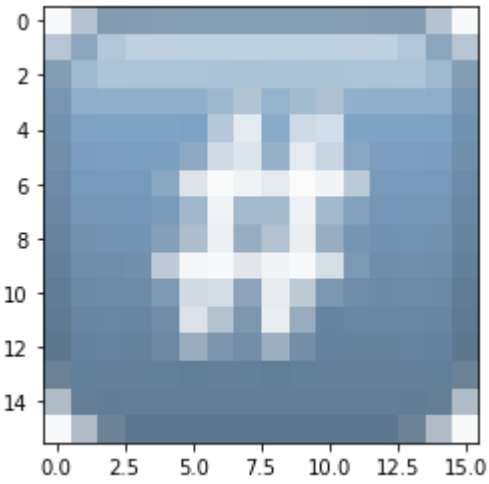
Q2a — Build a set of inputs $\boldsymbol{X}$ from the sheet images [5 marks]

Write code to build a Numpy array of inputs $\boldsymbol{X}$ having dtype *float32* and shape (N, D) where N is the total number of emoji images (number of emojis $\times$ number of vendors) and D is the total number of pixels per emoji (height $\times$ width $\times$ channels). Each image should be converted from RGBA (4 channels) to RGB (3 channels) using the *rgba_to_rgb* function defined at the top of this lab. The first rows of $\boldsymbol{X}$ should all be apple emojis, followed by all facebook emojis, then all google emojis, and finally the last rows should be all twitter emojis. **Briefly comment each non-trivial line of your code.**

In [32]:
```
# Your answer here. Aim for 6-10 lines.
emojiIndexes = [] #store all possitble (x,y) tuple into a list to get the emoji b
categories = []  #store each corresponding category into a list to initialize the
#names = []
for index in data:
    x_pos = index['sheet_x']   #parse the sheet_x to find x index
    y_pos = index['sheet_y']   #parse the sheet_y to find y index
    emojiIndexes.append((x_pos,y_pos))
    categories.append(index['category'])   #parse the category to find category c
    #names.append(index['name'])
X_list = []
for emojiIndex in range(0,len(emojiIndexes)):
    for style_index in range(0,4):                  #there are 4 styles of emojis
        # get the images with get_emoji_image method for every single emoji image
        rgba_emoji = rgba_to_rgb(get_emoji_image(style_index,emojiIndexes[emojiIn
        # append the rbga data of each image into X_list to then convert it to nu
        X_list.append(rgba_emoji)
X = np.array(X_list)   #convert the X_list into a np array and assign it as our X
X = X.reshape(-1,16*16*3) #shape it into 2 dimensions to train later on
```

Plot a row of your $\boldsymbol{X}$. Demonstrate to the TA that the rows of your $\boldsymbol{X}$ matrix is an emoji image. Do so by accessing a single row of $\boldsymbol{X}$ and plotting it as a 16×16 RGB image.

In [34]:
```
# Your answer here. Aim for 1-2 lines.
plt.imshow(X.reshape(-1,16,16,3)[0]);
```



Q2b — Build a set of targets y from the metadata [5 marks]

Here you'll enumerate the distinct emoji categories, and then build a vector of integer targets $\boldsymbol{y}$.

Write code to get a list of distinct emoji categories, using the *category* field from the metadata. For the TA, ensure that your list of categories is also displayed when the code cell below is executed.

In [35]:
```
# Your answere here. Aim for 2-4 lines.
print(categories[0:50])
#the categories that match the X without considering the 4 different styles of em
# is already initialized in the Q2a when parsing the json data of sheet_x and she
```

```
['Symbols', 'Symbols', 'Symbols', 'Symbols', 'Symbols', 'Symbols', 'Symbols',
'Symbols', 'Symbols', 'Symbols', 'Symbols', 'Symbols', 'Symbols', 'Symbols', 'A
ctivities', 'Activities', 'Symbols', 'Symbols', 'Symbols', 'Symbols', 'Symbol
s', 'Symbols', 'Symbols', 'Symbols', 'Symbols', 'Symbols', 'Symbols', 'Symbol
s', 'Symbols', 'Symbols', 'Symbols', 'Flags', 'Flags', 'Flags', 'Flags', 'Flag
s', 'Flags', 'Flags', 'Flags', 'Flags', 'Flags', 'Flags', 'Flags', 'Flags', 'Fl
ags', 'Flags', 'Flags', 'Flags', 'Flags', 'Flags']
```

Write code to build a Numpy array of inputs $\boldsymbol{y}$ having dtype *int32* and where $y_i \in \{0, \ldots, M-1\}$ with M being the number of distinct emoji categories. The order of items in $\boldsymbol{y}$ should match those of $\boldsymbol{X}$ from **Q2a**. You may use any approach you like, but potentially useful functions include the *list* object's index (https://docs.python.org/3/tutorial/datastructures.html) function and Numpy's np.tile (https://numpy.org/doc/stable/reference/generated/numpy.tile.html) function. **Briefly comment each non-trivial line of your code.**

In [36]:
```
# Your answer here. Aim for 3-5 lines.
y = []
for c in categories:
    for x in range(0,4):  #since the category for each style (apple,twitter...etc
        y.append(c)      # so append them into list 4 times
y = np.array(y)         #convert y into numpy array
```

Write code to demonstrate that, for each training example i corresponding to a '*laughing*' emoji (for apple, facebook, google, twitter), its y_i label is set to be the index of the "Smileys & Emotion" category.

In [37]:
```
# Your answer here. Aim for 1-3 lines.
counter = 0
for i in range(0,len(y)):
    if counter == 10:    #printing 10 images with category correspond to 'Smileys
        break
    if(y[i] == 'Smileys & Emotion'):
        plt.figure()
        plt.imshow(X.reshape(-1,16,16,3)[i])
        plt.title(y[i])
        counter +=1
```

Q2c — Split and preprocess the data [10 marks]

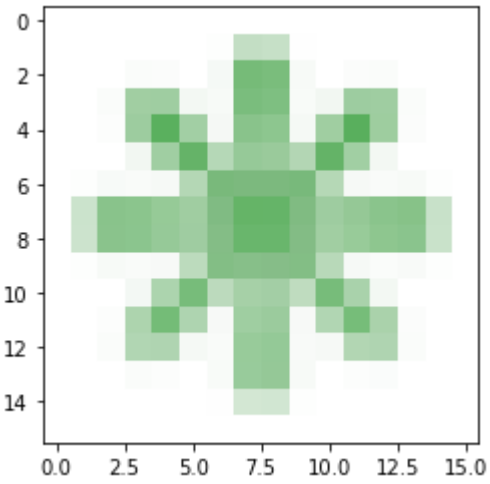
Write code to randomly split $(\boldsymbol{X}, \boldsymbol{y})$ into three parts, with no overlap:

1. a *training* set $(\boldsymbol{X}_{\text{trn}}, \boldsymbol{y}_{\text{trn}})$, which you will use to directly train classifiers
2. a *validation* set $(\boldsymbol{X}_{\text{val}}, \boldsymbol{y}_{\text{val}})$, which you will use to estimate the best value for a hyperparameter
3. a *test* set $(\boldsymbol{X}_{\text{tst}}, \boldsymbol{y}_{\text{tst}})$, which you will use to evaluate final performance of the 'best' hyperparameters

The training data should comprise 60% of the full data set. The validation and testing data should each comprise 20% of the original data. Use the [train_test_split (https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.train_test_split.html)](https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.train_test_split.html) function and remember to set *random_state* so that your splits (and thereby your conclusions) are reproducible for TAs.

In [79]:
```
# Your code here. Aim for 2-3 lines.
# train = 60%
# test = 20%
# val = 20%
X_trn, X_tst, y_trn, y_tst = sklearn.model_selection.train_test_split(X, y, test_
X_trn, X_val, y_trn, y_val = sklearn.model_selection.train_test_split(X_trn, y_tr
plt.imshow(X_val.reshape(-1,16,16,3)[0])
```

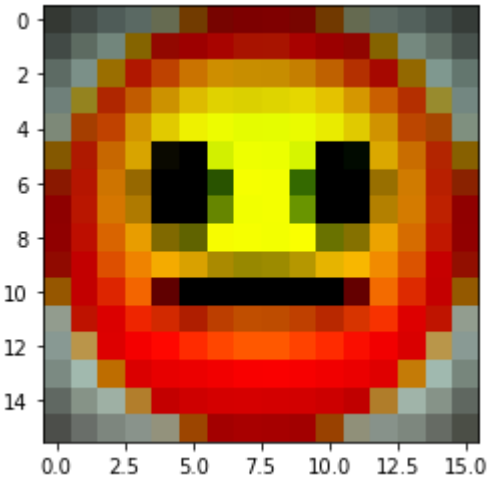
Out[79]:
```
<matplotlib.image.AxesImage at 0x22b3c6d0070>
```



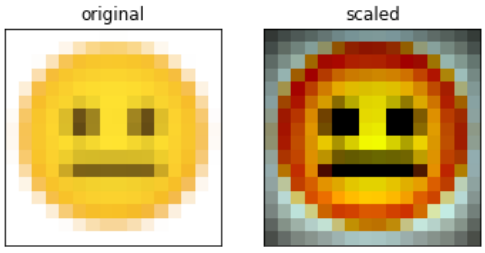
Write code to normalize the features of $\boldsymbol{X}_{\text{trn}}$, $\boldsymbol{X}_{\text{val}}$, and $\boldsymbol{X}_{\text{tst}}$, using scikit-learn's [StandardScaler (https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html)](https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html). Be careful which subset of the data you use for estimating the *StandardScaler* object's *scale_* and *mean_* attributes.

In [66]:
```
# Your answer here. Aim for 4-5 lines.
scaler = sklearn.preprocessing.StandardScaler()
def applyScale(X_data):
    newX = scaler.fit_transform(X_data)
    return newX

X_trn = applyScale(X_trn)
X_val = applyScale(X_val)
X_tst = applyScale(X_tst)

plt.imshow(X_tst.reshape(-1,16,16,3)[37]);
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for flo
ats or [0..255] for integers).
```

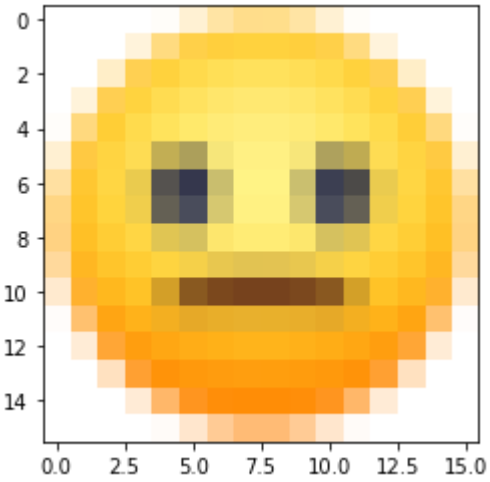


Plot a scaled and unscaled emoji side-by-side. Choose a row from $\boldsymbol{X}_{\text{trn}}$ and show how it appears as an RGB image with and without scaling. If Matplotlib complains that "values are not in range [0,1]," consider using [np.clip (https://numpy.org/doc/stable/reference/generated/numpy.clip.html)](https://numpy.org/doc/stable/reference/generated/numpy.clip.html). Your plot should look similar to the example below, although the choice of emoji could differ.



Hint: You do not need to know which row in $\boldsymbol{X}_{\text{trn}}$ corresponds to which row in $\boldsymbol{X}$. Instead you can "undo" the scaling on whatever row you pick, using one of the methods provided by *StandardScaler*.

In [67]:
```
# Your answer here. Aim for 9-12 lines.
def unapplyScale(X_data):
    newX = scaler.inverse_transform(X_data)
    return newX

X_tst = unapplyScale(X_tst)
plt.imshow(X_tst.reshape(-1,16,16,3)[37]);
```



Q3 — Train classifiers and identify good hyperparameters [20 marks total]

This question has several goals:

1. to help you visualize how hyperparameters affect training/validation/test performance.
2. to give you a sense for how long certain classifiers take to train or to make predictions.
3. to force you to try two useful Python language features: (a) passing types as arguments, and (b) argument forwarding.

(However, please take the hyperparameter search lab as a better example of how to use scikit-learn for hyperparameter search; this assignment is focused on making things easy to plot and visualize, not on automating the search itself.)

Q3a — Write a utility function that trains multiple estimators [5 marks]

Throughout **Q3**, you will be training multiple estimators, each with a different hyperparameter setting.

Implement the *train_estimators* utility function. The idea of this function is to make it easy to train multiple versions of an estimator where a single hyperparameter (specified by *param_name*) takes on a different value (specified by *param_vals*) for each estimator. See the docstring below.

Hint: For details on how Python argument forwarding works (`**kwargs`), see [this Stack Overflow answer (https://stackoverflow.com/a/36908)](https://stackoverflow.com/a/36908).

In [48]:
```
def train_estimators(X, y, estimator_type, param_name, param_vals, **kwargs):
    """
    Trains multiple instances of `estimator_type` on (X, y) by setting argument
    named `param_name` to each value in `param_vals`. Prints a message before
    training each instance. Returns the list of trained estimators.

    For example:

       >>> train_estimators(X, y, DecisionTreeClassifier, 'max_depth', [1, 5, 10]
                            splitter='random', random_state=0)

       Training DecisionTreeClassifier(max_depth=1, random_state=0, splitter='ran
       Training DecisionTreeClassifier(max_depth=5, random_state=0, splitter='ran
       Training DecisionTreeClassifier(max_depth=10, random_state=0, splitter='ra

       [DecisionTreeClassifier(max_depth=1, random_state=0, splitter='random'),
        DecisionTreeClassifier(max_depth=5, random_state=0, splitter='random'),
        DecisionTreeClassifier(max_depth=10, random_state=0, splitter='random')]
    """
    # Your implementation here. Aim for 5-10 lines.
    trainned_estimators = []
    for i in range(0,len(param_vals)):
        clf = estimator_type(**kwargs)
        clf.set_params(**{param_name:param_vals[i]})
        clf.fit(X,y)
        trainned_estimators.append(clf)
        print('Training ', clf, '...')
    return trainned_estimators
```

Run the code cell below to test your implementation of *train_estimators*. (Replace *X_trn* and *y_trn* with whatever you called your training set variables.)

In [49]:
```
tree_estimators = train_estimators(X_trn, y_trn, sklearn.tree.DecisionTreeClassif
                                   'max_depth', [1, 5, 10], splitter='random', ra
tree_estimators
```

```
Training  DecisionTreeClassifier(max_depth=1, random_state=0, splitter='rando
m') ...
Training  DecisionTreeClassifier(max_depth=5, random_state=0, splitter='rando
m') ...
Training  DecisionTreeClassifier(max_depth=10, random_state=0, splitter='rando
m') ...
```

Out[49]:
```
[DecisionTreeClassifier(max_depth=1, random_state=0, splitter='random'),
 DecisionTreeClassifier(max_depth=5, random_state=0, splitter='random'),
 DecisionTreeClassifier(max_depth=10, random_state=0, splitter='random')]
```

Q3b — Train multiples models, plot their accuracies, and identify good parameters [15 marks]

Implement the *score_estimators* utility function. This will be handy for scoring a list of estimators on a particular data set, such as $(\boldsymbol{X}_{\text{trn}}, \boldsymbol{y}_{\text{trn}})$. Use the estimator's own *score* method.

In [50]:
```
def score_estimators(X, y, estimators):
    """Scores each estimator on (X, y), returning a list of scores."""
    # Your implementation here. Aim for 1-4 lines.
    accuracy_scores = []
    for i in range(0, len(estimators)):
        accuracy_scores.append(roundVal(estimators[i].score(X,y)))
    return accuracy_scores
```

Run the code cell below to test your implementation. It should print three scores per dataset. Each score is a measure of classification accuracy. (Replace *X_trn* and *y_trn* etc with your dataset variable names.)

In [51]:
```
print("train:   ", score_estimators(X_trn, y_trn, tree_estimators))
print("validate:", score_estimators(X_val, y_val, tree_estimators))
print("test:    ", score_estimators(X_tst, y_tst, tree_estimators))
```

```
train:    [0.321, 0.535, 0.779]
validate: [0.285, 0.511, 0.557]
test:     [0.292, 0.507, 0.534]
```

You should see that the 3rd column (corresponding to *max_depth=15*) performs best, especially on training.

Run the code cell below to see a demonstration of the `%%time` feature of Jupyter (see here (https://ipython.readthedocs.io/en/stable/interactive/magics.html#magic-time)). Note that `%%time` only works if it is the first line in a code cell, before any comments.

In [52]:
```
%%time
for i in range(1000000):  # Burn some CPU cycles in a
    pass                  # loop that does nothing
```

```
Wall time: 42 ms
```

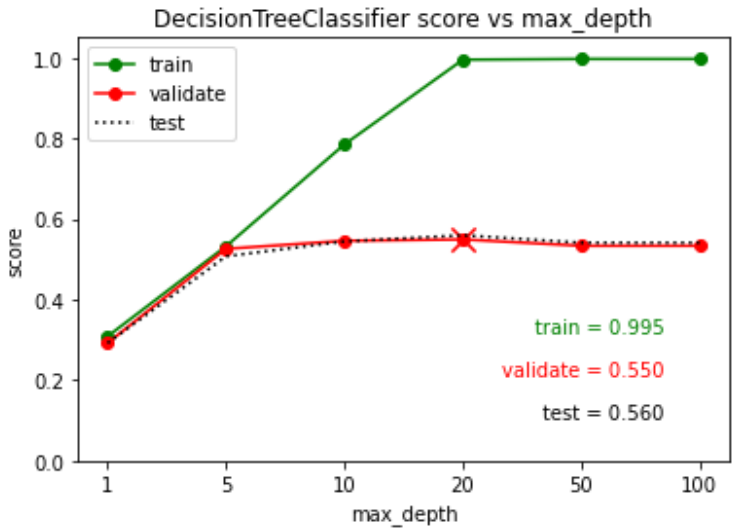
Train multiple *DecisionTreeClassifier*s on the training set, such that *train_estimators* produces the following output:

```
Training DecisionTreeClassifier(max_depth=1, random_state=0, splitter='r
andom')...
Training DecisionTreeClassifier(max_depth=5, random_state=0, splitter='r
andom')...
Training DecisionTreeClassifier(max_depth=10, random_state=0, splitter
='random')...
Training DecisionTreeClassifier(max_depth=20, random_state=0, splitter
='random')...
Training DecisionTreeClassifier(max_depth=50, random_state=0, splitter
='random')...
Training DecisionTreeClassifier(max_depth=100, random_state=0, splitter
='random')...
```

In [53]:
```
%%time
# Your answer here. Aim for 1-2 lines.
tree_estimators = train_estimators(X_trn, y_trn, sklearn.tree.DecisionTreeClassif
                                   'max_depth', [1, 5, 10, 20, 50, 100], splitter
```

```
Training  DecisionTreeClassifier(max_depth=1, random_state=0, splitter='rando
m') ...
Training  DecisionTreeClassifier(max_depth=5, random_state=0, splitter='rando
m') ...
Training  DecisionTreeClassifier(max_depth=10, random_state=0, splitter='rando
m') ...
Training  DecisionTreeClassifier(max_depth=20, random_state=0, splitter='rando
m') ...
Training  DecisionTreeClassifier(max_depth=50, random_state=0, splitter='rando
m') ...
Training  DecisionTreeClassifier(max_depth=100, random_state=0, splitter='rando
m') ...
Wall time: 1.66 s
```

Implement *plot_estimator_scores* to visualize the effect of the parameter on accuracy. When applied to the decision tree estimators you trained in the previous cell, the plot should look like below, including legend, colours, marks, and x-axis ticks, but your precise scores may differ depending on how you decided to split the data.



Hint: You can use your *score_estimators* implementation, but do not do any training.

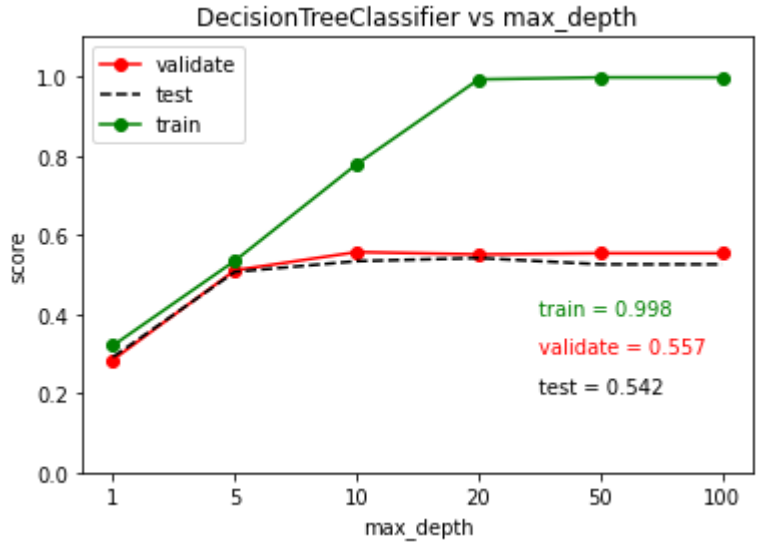
Hint: For the title, you can get the object's type from its `__class__` attribute, and you can get the name of its type from the type's `__name__` attribute. Use the first object in *estimators* to determine the name of the classifier type that you're plotting.

Hint: If your x-axis points are not evenly spaced, you can plot each series using any evenly-spaced x values (e.g. via *np.arange*) and then override the x-axis tick labels with whatever you want. See the *labels* argument of Matplotlib's **xticks (https://matplotlib.org/3.1.1/api/_as_gen/matplotlib.pyplot.xticks.html)** function.

In [54]:
```
def plot_estimator_scores(estimators, param_name, param_vals):
    """
    Plots the training, validation, and testing scores of a list of estimators,
    where `param_name` and `param_vals` are the same as for `train_estimators`.
    The estimator with best validation score will be highlighted with an 'x'.
    """
    # Your implementation here. Use as many lines as you need.
    x_arange = np.arange(len(param_vals))

    # compute the score of each data and store them as separate list
    y_ax_trn = score_estimators(X_trn, y_trn, estimators)
    y_ax_tst = score_estimators(X_tst, y_tst, estimators)
    y_ax_val = score_estimators(X_val, y_val, estimators)

    # plot each of the ata
    plt.plot(x_arange, y_ax_val, color='red', marker='o', label = "validate");
    plt.plot(x_arange, y_ax_tst, color='black', linestyle = 'dashed', label = "te
    plt.plot(x_arange, y_ax_trn, color='green', marker='o', label = "train");
    plt.xticks(x_arange,param_vals)

    plt.xlabel(param_name)
    plt.ylabel('score')
    plt.title(str(estimators[0].__class__.__name__) + ' vs ' + param_name)
    plt.legend()
    plt.text(3.5, 0.4, 'train = ' + str(max(y_ax_trn)) , fontsize=10, color = 'gr
    plt.text(3.5, 0.3, 'validate = '+ str(max(y_ax_val)), fontsize=10, color = 'r
    plt.text(3.5, 0.2, 'test = ' + str(max(y_ax_tst)), fontsize=10, color = 'blac
    plt.ylim([0,1.1])
    plt.show()
```

Plot the *DecisionTreeClassifier* scores by calling your *plot_estimator_scores* function. Your plot should look like the example plot.

In [55]:
```
%%time
# Your code here. Aim for 1 line.
plot_estimator_scores(tree_estimators, 'max_depth', [1, 5, 10, 20, 50, 100])
```




```
Wall time: 266 ms
```

Train multiple *RandomForestClassifiers* such that *train_estimators* produces the following output:

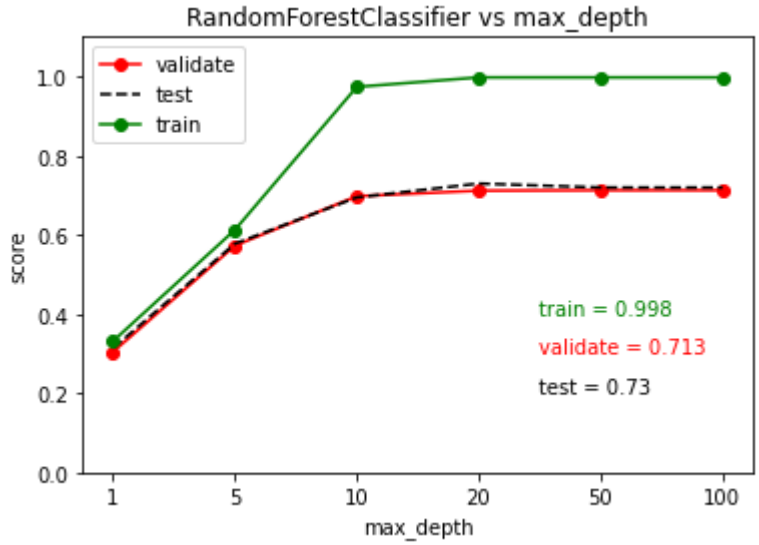
```
Training RandomForestClassifier(max_depth=1, random_state=0)...
Training RandomForestClassifier(max_depth=5, random_state=0)...
Training RandomForestClassifier(max_depth=10, random_state=0)...
Training RandomForestClassifier(max_depth=20, random_state=0)...
Training RandomForestClassifier(max_depth=50, random_state=0)...
Training RandomForestClassifier(max_depth=100, random_state=0)...
```

In [56]:
```
%%time
# Your code here. Aim for 1-2 lines.
randomForest_estimators = train_estimators(X_trn, y_trn, sklearn.ensemble.RandomF
                                           'max_depth', [1, 5, 10, 20, 50, 100],  random_
```

```
Training  RandomForestClassifier(max_depth=1, random_state=0) ...
Training  RandomForestClassifier(max_depth=5, random_state=0) ...
Training  RandomForestClassifier(max_depth=10, random_state=0) ...
Training  RandomForestClassifier(max_depth=20, random_state=0) ...
Training  RandomForestClassifier(max_depth=50, random_state=0) ...
Training  RandomForestClassifier(max_depth=100, random_state=0) ...
Wall time: 22.9 s
```

Plot the *RandomForestClassifier* scores, again by calling your *plot_estimator_scores* function.

In [57]:
```
%%time
# Your code here. Aim for 1 line.
plot_estimator_scores(randomForest_estimators, 'max_depth', [1, 5, 10, 20, 50, 10
```




```
Wall time: 1.27 s
```

Train multiple *LogisticRegression* classifiers such that *train_estimators* produces the following output:

```
Training LogisticRegression(C=1e-05, max_iter=10000, random_state=0)...
Training LogisticRegression(C=0.0001, max_iter=10000, random_state=0)...
Training LogisticRegression(C=0.001, max_iter=10000, random_state=0)...
Training LogisticRegression(C=0.01, max_iter=10000, random_state=0)...
Training LogisticRegression(C=0.1, max_iter=10000, random_state=0)...
Training LogisticRegression(max_iter=10000, random_state=0)...
```

The omission of *C* when the final estimator was printed means it was trained with its default value, which is *C*=1. You can try it yourself:

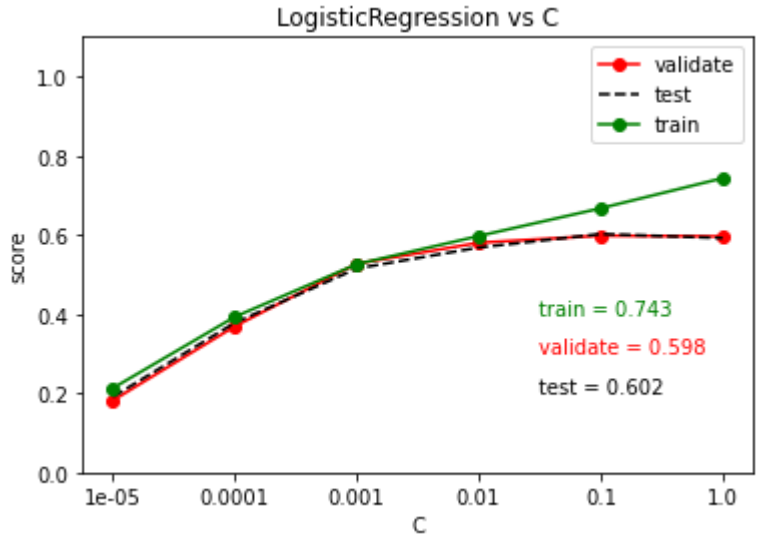
```
>>> LogisticRegression(C=1.01)
LogisticRegression(C=1.01)

>>> LogisticRegression(C=1.0)
LogisticRegression()
```

In [75]:
```
%%time
# Your code here. Aim for 1-2 lines
log_estimators = train_estimators(X_trn, y_trn, sklearn.linear_model.LogisticRegr
                                  'C', [0.00001, 0.0001, 0.001, 0.01, 0.1, 1.0],
```

```
Training  LogisticRegression(C=1e-05, max_iter=10000, random_state=0) ...
Training  LogisticRegression(C=0.0001, max_iter=10000, random_state=0) ...
Training  LogisticRegression(C=0.001, max_iter=10000, random_state=0) ...
Training  LogisticRegression(C=0.01, max_iter=10000, random_state=0) ...
Training  LogisticRegression(C=0.1, max_iter=10000, random_state=0) ...
Training  LogisticRegression(max_iter=10000, random_state=0) ...
Wall time: 1min 36s
```

Plot the *LogisticRegression* scores, again by calling your *plot_estimator_scores* function.

In [76]:
```
%%time
# Your code here. Aim for 1 line.
plot_estimator_scores(log_estimators, 'C', [0.00001, 0.0001, 0.001, 0.01, 0.1, 1.
```




```
Wall time: 410 ms
```

Train multiple SVM classifiers (*SVC*) such that *train_estimators* produces the following output:

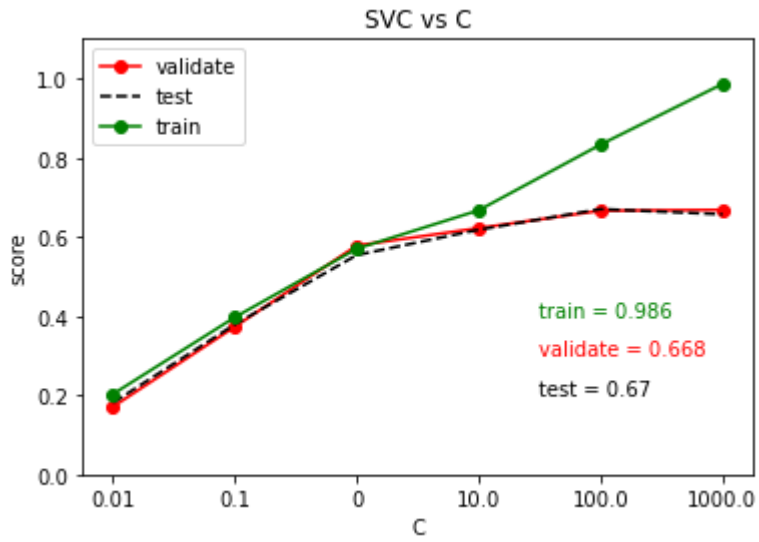
```
Training SVC(C=0.01, gamma=0.001, max_iter=10000, random_state=0)...
Training SVC(C=0.1, gamma=0.001, max_iter=10000, random_state=0)...
Training SVC(gamma=0.001, max_iter=10000, random_state=0)...
Training SVC(C=10.0, gamma=0.001, max_iter=10000, random_state=0)...
Training SVC(C=100.0, gamma=0.001, max_iter=10000, random_state=0)...
Training SVC(C=1000.0, gamma=0.001, max_iter=10000, random_state=0)...
```

In [71]:
```
%%time
# Your code here. Aim for 1-2 lines.
svm_estimators = train_estimators(X_trn, y_trn, sklearn.svm.SVC,
                                  'C', [0.01, 0.1, 1, 10.0, 100.0, 1000.0], gamm
```

```
Training  SVC(C=0.01, gamma=0.001, max_iter=10000, random_state=0) ...
Training  SVC(C=0.1, gamma=0.001, max_iter=10000, random_state=0) ...
Training  SVC(C=1, gamma=0.001, max_iter=10000, random_state=0) ...
Training  SVC(C=10.0, gamma=0.001, max_iter=10000, random_state=0) ...
Training  SVC(C=100.0, gamma=0.001, max_iter=10000, random_state=0) ...
Training  SVC(C=1000.0, gamma=0.001, max_iter=10000, random_state=0) ...
Wall time: 37.8 s

C:\Users\William\anaconda3\lib\site-packages\sklearn\svm\_base.py:255: Converge
nceWarning: Solver terminated early (max_iter=10000).  Consider pre-processing
your data with StandardScaler or MinMaxScaler.
  warnings.warn('Solver terminated early (max_iter=%i).'
```

Plot the *SVM* scores, again by calling your *plot_estimator_scores* function. Predictions may take several minutes to compute.

In [73]:
```
%%time
# Your code here. Aim for 1 line.
plot_estimator_scores(svm_estimators, 'C', [0.01, 0.1, 0, 10.0, 100.0, 1000.0])
```




```
Wall time: 1min 29s
```

Question. Do your plots support the claim that "validation set performance" is a good estimate of "test set performance" overall? YES/NO then explain below.

```
Your answer here
Yes, validation set is a good estimate of test set performance. We can observe
that our validation graph is very similar to our test graph showing. It is
good plot to estimate whether our trainning may be under-fitted or over-
fitted, and also it can be viewed as a test set in case our testing data is
not well chosen to be a good testing sets that estimate this learning
algorithm.
```

Question. Which of your classifiers had the highest test-set performance for its "best" configuration (i.e., for the configuration with highest validation-set performance)? Name the classifier and best hyperparameter setting (*max_depth* or *C*).

```
Your answer here
The random forest classifier with max_depth = 20 has the highest test-set
performance: 0.73.
```

Question. Which of your classifiers had the *least over-fitting*, if we measured overfitting as the absolute difference between training-set and testing-set performance? Name the classifier and hyperparameter setting (*max_depth* or *C*).

```
Your answer here
the logistic regression model has the the least-over-fitting since the
difference between the testing line and the trainning line is almost equal to
0 throughout the graph. only when C is > 0.01, the absolute difference between
training-set and testing-set become slightly larger.
```

Question. Which if your classifiers was slowest to train? Name the classifier.

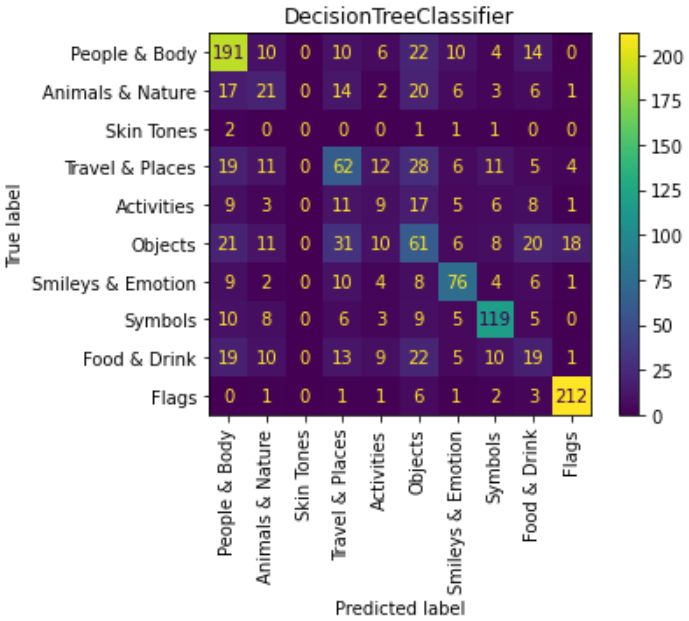
```
Your answer here
the logistic regression model was the slowest to train with a training time of
1min 36s.
```

Q4 — Visualizing mistakes [10 marks total]

The goal here is to visualize classification errors, by confusion matrix and by inspecting typical mistakes.

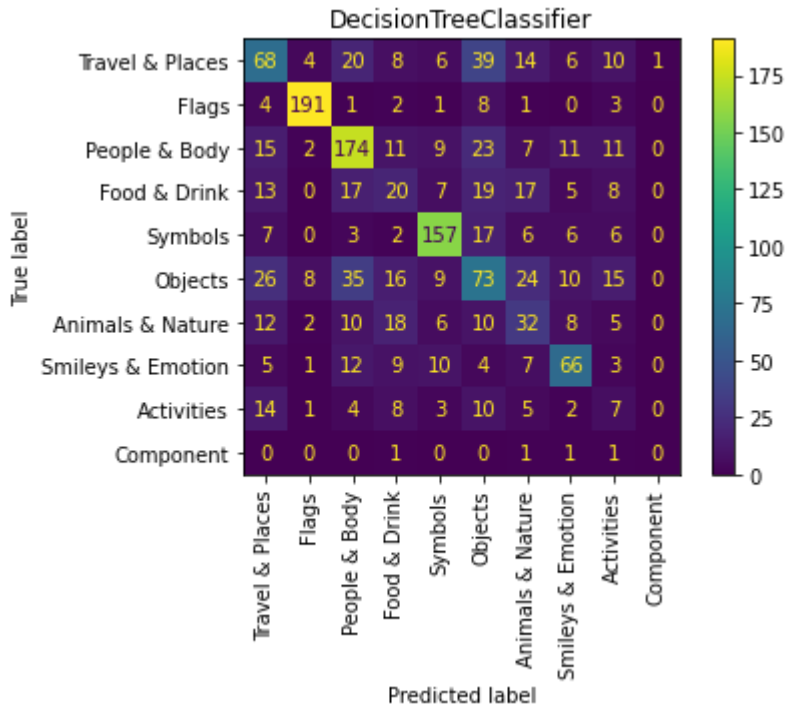
Q4a — Plot a confusion matrix for the best estimators [5 marks]

Write code to plot a confusion matrix for each of the 'best' estimators in **Q3** when applied to test set $(\boldsymbol{X}_{\text{tst}}, \boldsymbol{y}_{\text{tst}})$. Here, 'best' means best validation score. All estimators are already trained, so you can simply pull out the one best of each type {tree, forest, logistic, svm}. Use [plot_confusion_matrix (https://scikit-learn.org/stable/modules/generated/sklearn.metrics.plot_confusion_matrix.html)](https://scikit-learn.org/stable/modules/generated/sklearn.metrics.plot_confusion_matrix.html) or [ConfusionMatrixDisplay (https://scikit-learn.org/stable/modules/generated/sklearn.metrics.ConfusionMatrixDisplay.html)](https://scikit-learn.org/stable/modules/generated/sklearn.metrics.ConfusionMatrixDisplay.html). Your first plot should look like as below, though the numbers may differ.

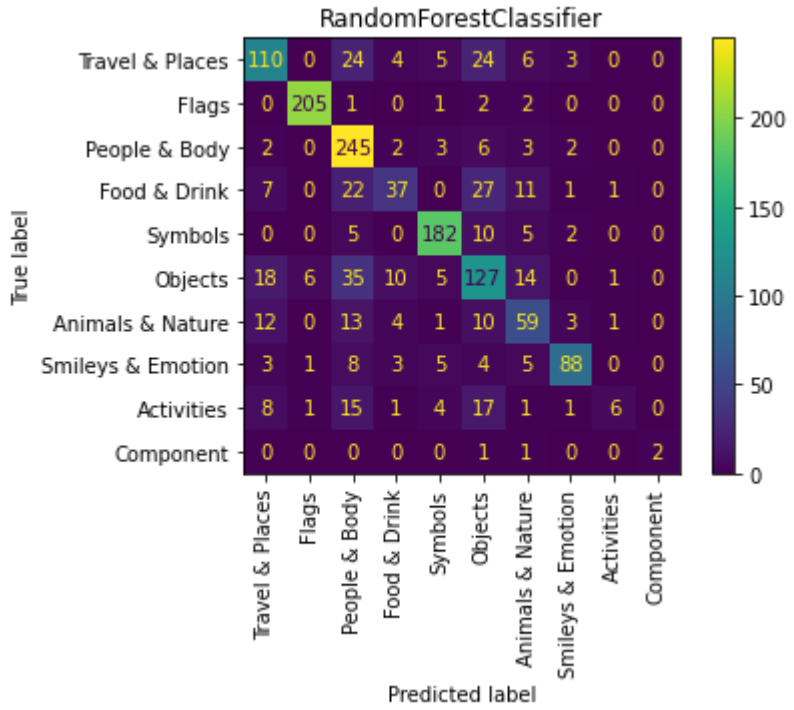

DecisionTreeClassifier
People & Body 191 10 0 10 6 22 10 4 14 0
Animals & Nature 17 21 0 14 2 20 6 3 6 1
Skin Tones 2 0 0 0 0 1 1 1 0 0
Travel & Places 19 11 0 62 12 28 6 11 5 4
Activities 9 3 0 11 9 17 5 6 8 1
Objects 21 11 0 31 10 61 6 8 20 18
Smileys & Emotion 9 2 0 10 4 8 76 4 6 1
Symbols 10 8 0 6 3 9 5 119 5 0
Food & Drink 19 10 0 13 9 22 5 10 19 1
Flags 0 1 0 1 1 6 1 2 3 212
True label
People & Body
Animals & Nature
Skin Tones
Travel & Places
Activities
Objects
Smileys & Emotion
Symbols
Food & Drink
Flags
Predicted label
200
175
150
125
100
75
50
25
0

In [77]:
```
# Your answer here. Aim for 7-12 lines.
def plotConfusionMatrix(estimators, X, y):
    plt.figure()
    predictions = estimators[3].predict(X)
    categories_noDup = list(dict.fromkeys(y))
    matr = sklearn.metrics.confusion_matrix(y, predictions, labels= categories_nc
    disp = sklearn.metrics.ConfusionMatrixDisplay(confusion_matrix=matr, display_
    disp.plot(xticks_rotation='vertical')
    plt.xlabel('Predicted label')
    plt.ylabel('True label')
    plt.title(str(estimators[0].__class__.__name__));
    return matr

plotConfusionMatrix(tree_estimators,X_tst,y_tst)
plotConfusionMatrix(randomForest_estimators,X_tst,y_tst)
plotConfusionMatrix(svm_estimators,X_tst,y_tst)
cm = plotConfusionMatrix(log_estimators,X_tst,y_tst)
```

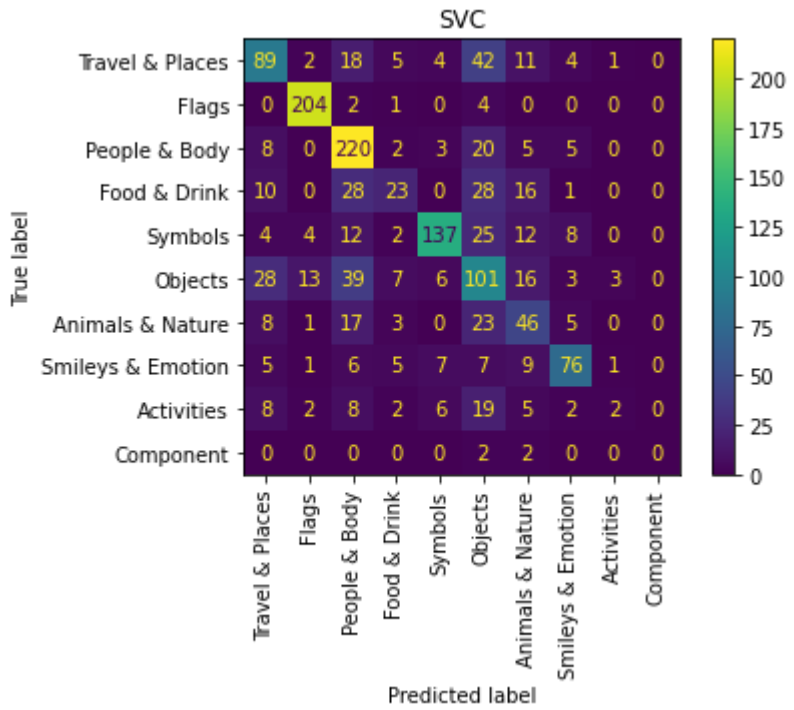
```
<Figure size 432x288 with 0 Axes>
```



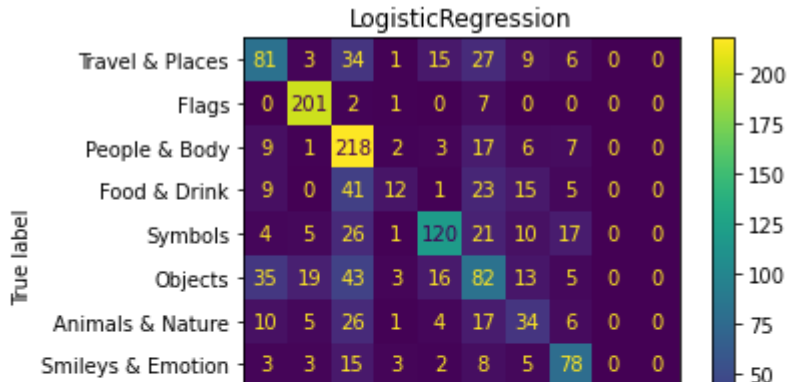

```
<Figure size 432x288 with 0 Axes>
```



<Figure size 432x288 with 0 Axes>



<Figure size 432x288 with 0 Axes>



Question. What classifier is best at distinguishing between the *Flags* class and the *Objects* class, overall? Name the classifier and justify your choice.

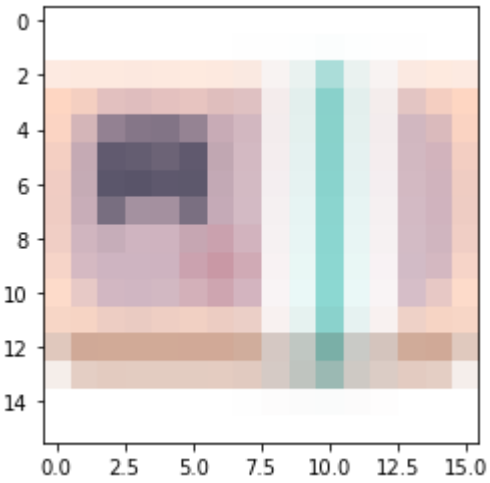
```
Your answer here
random forest classifier is the best at distinguishing flags and objects
because out of 211 emojis, it has successfully identified 205/211 flags and
only missclassified 6/211 of flags. In addition, it has predicted 127/129
objects right while only classfied 2/129 of objects into flags
```

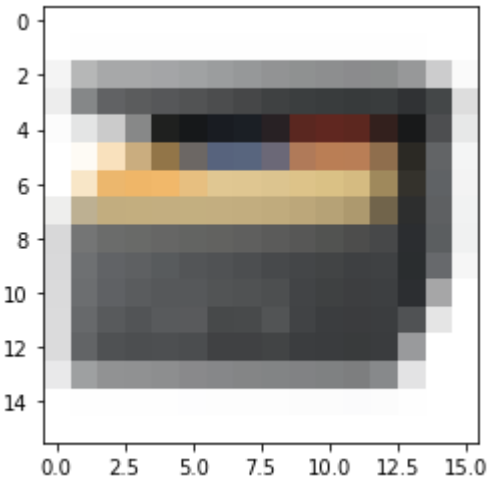
Q4b — Identify specific misclassified examples [5 marks]

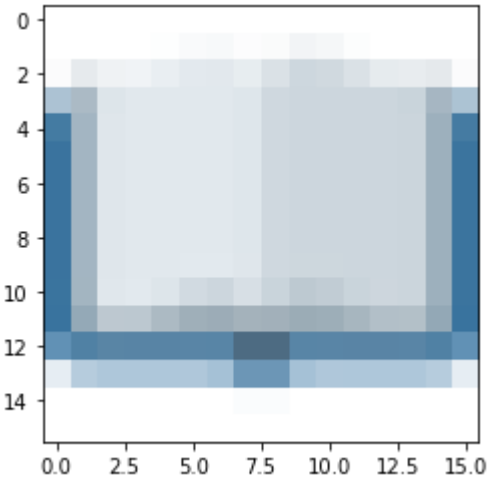
In this question, use the "best performing" classifier that you named at the end of **Q4a**.

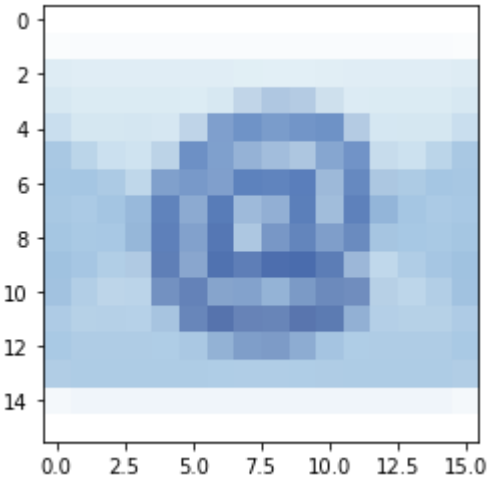
Write code to identify all "*Objects* misclassified as *Flags*" from the test-set and then plot them as images. (The phrase "*A* misclassified as *B*" means the prediction was *B* but the true class was *A*.) Your code for identifying the misclassified examples should be vectorized, for example using functions like *np.logical_and* and/or *np.nonzero*. (Remember you might need to "undo" the feature normalization like in **Q2c**).

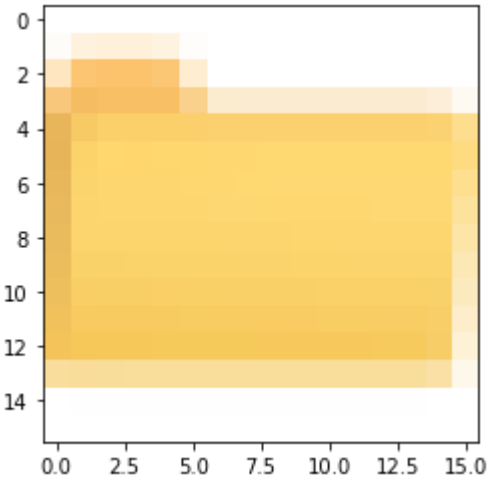
In [116]:
```
# Your answer here. Aim for 7-12 lines.
predicted_tst = randomForest_estimators[3].predict(X_tst)
misclassifieds = []
for i in range(0,len(y_tst)):
    if y_tst[i] == 'Objects' and predicted_tst[i] == 'Flags':
        misclassifieds.append(i)
for x in range(0,len(misclassifieds)):
    plt.figure()
    plt.imshow(X_tst.reshape(-1,16,16,3)[misclassifieds[x]])
```

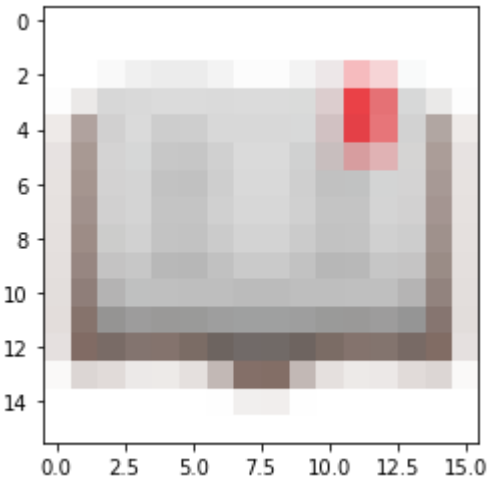












Question. After seeing the failure cases above, can you guess why the estimator is confusing them with *Flags*? Explain in 1-2 sentences.

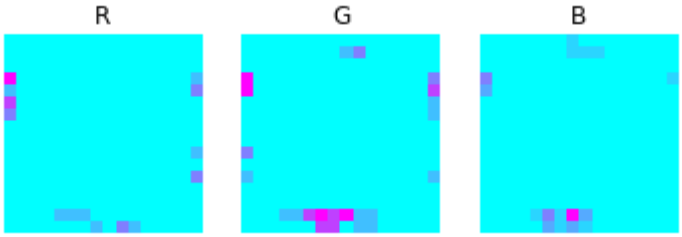
```
Your answer here
After seeing the failure cases above, I can see that the estimator is
confusing them because they are shaped rectangulars just like flags.
```

Q5 — Visualizing feature importances [10 marks total]

The goal here is to visualize sensitivity to specific input features.

Q5a — Visualize the feature importances of a RandomForestClassifier [5 marks]

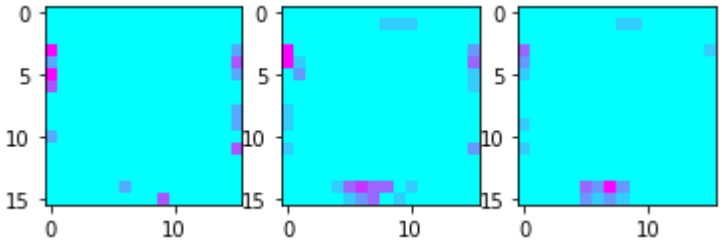
Implement *plot_random_forest_importances* below. This function should plot the *feature_importances_* attribute of a *RandomForestClassifier* (see scikit-learn docs). For the random forests you trained, there are $16 \times 16 \times 3$ features, so to make visualization easy the feature importances should be organized into three separate side-by-side heatmaps: one for each RGB colour channel. When plotting a heatmap, use *cmap*='cool' to choose the colour map. For example, plotting the feature importances of a random forest with *max_depth*=1 should look something like this:



In [214]:
```
def plot_random_forest_importances(estimator):
    """
    Plots the feature importances of the given RandomForestClassifier,
    arranged as three separate 16x16 heatmaps for (red, green, blue).
    """
    # Your implementation here. Aim for 7-10 lines.
    feature_importances = estimator.feature_importances_
    feature_importances = feature_importances.reshape(16,16,3)
    for j in range(0,3):
        plt.subplot(1,3,j+1)
        plt.imshow(feature_importances[:,:,j], cmap = 'cool')
```

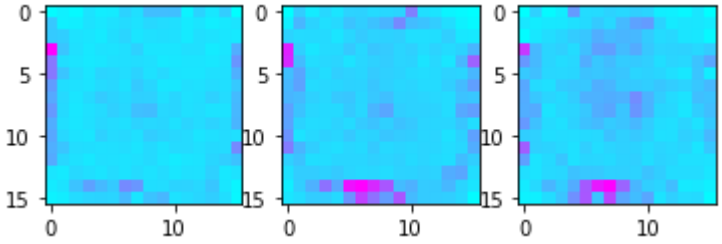
Check your code by calling your function to plot the feature importances of first *RandomForestClassifier* that you trained (with *max_depth*=1).

In [215]:
```
# Your code here (1 line)
plot_random_forest_importances(randomForest_estimators[0])
```



Plot the feature importances of your 'best' *RandomForestClassifier* instance. The patterns should be more complex.

In [216]:
```
# Your code here (1 line)
plot_random_forest_importances(randomForest_estimators[3])
```



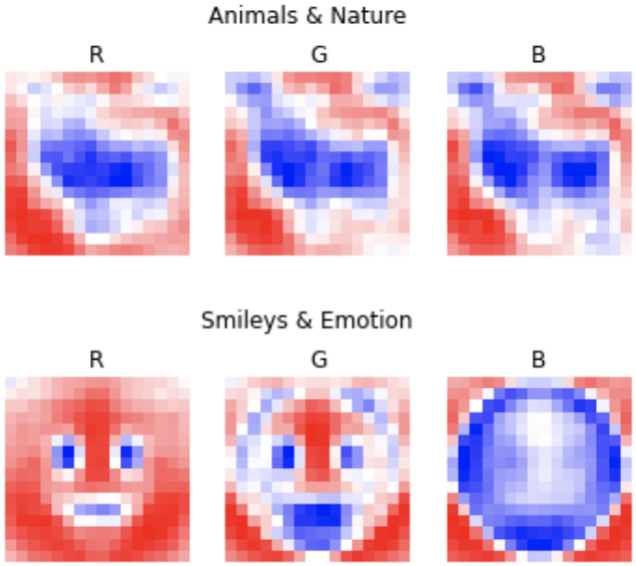
Question. Why do you think the features near the edge of the image so 'important'? Explain in 1-2 sentences.

```
Your answer here
Because features near the edge of the image determines the shape of the emoji.
For example, a flag is rectangular, it would have white space outside of
rectangle to indicate that it is a rectangular shape. Knowing the shape of an
image is crucial for any algorithm.
```

Q5b — Visualize the feature weights of LogisticRegression [5 marks]

This question is essentially the same as **Q5a** except you will extract the per-class weights of a *LogisticRegression* estimator that was trained on emoji images.

Implement *plot_logistic_weights* so that for each of the 10 categories of emoji it plots three side-by-side images. Use the *coef_* attribute of *LogisticRegression* to extract the $16 \times 16 \times 3$ weights for each category, and then generate a separate heatmap for each RGB channel. Since we want to see clearly which weights are positive or negative, use *cmap*='bwr' when plotting each heatmap. Use *suptitle* to label each group of heatmaps with its category label. Your function should generate 10×3 heatmaps total. Below are examples of 2 of the 10 possible rows:

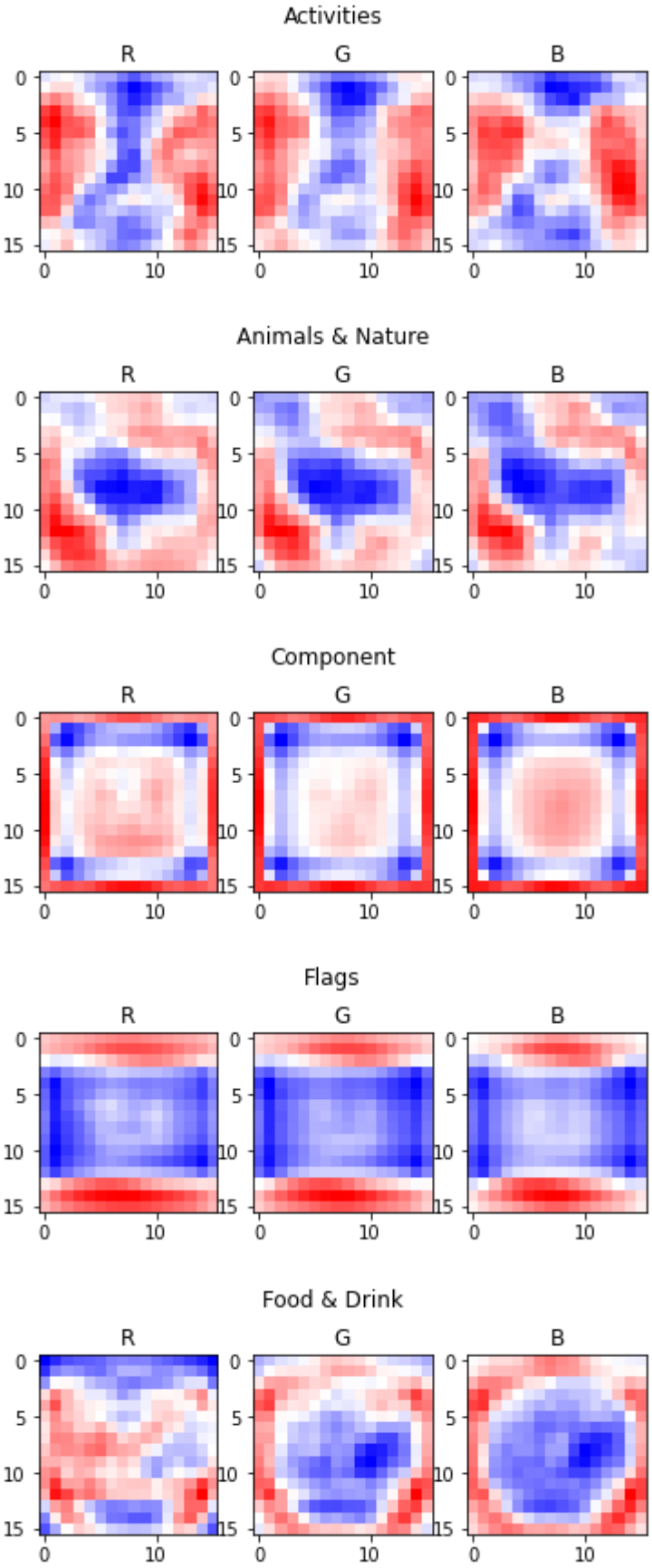


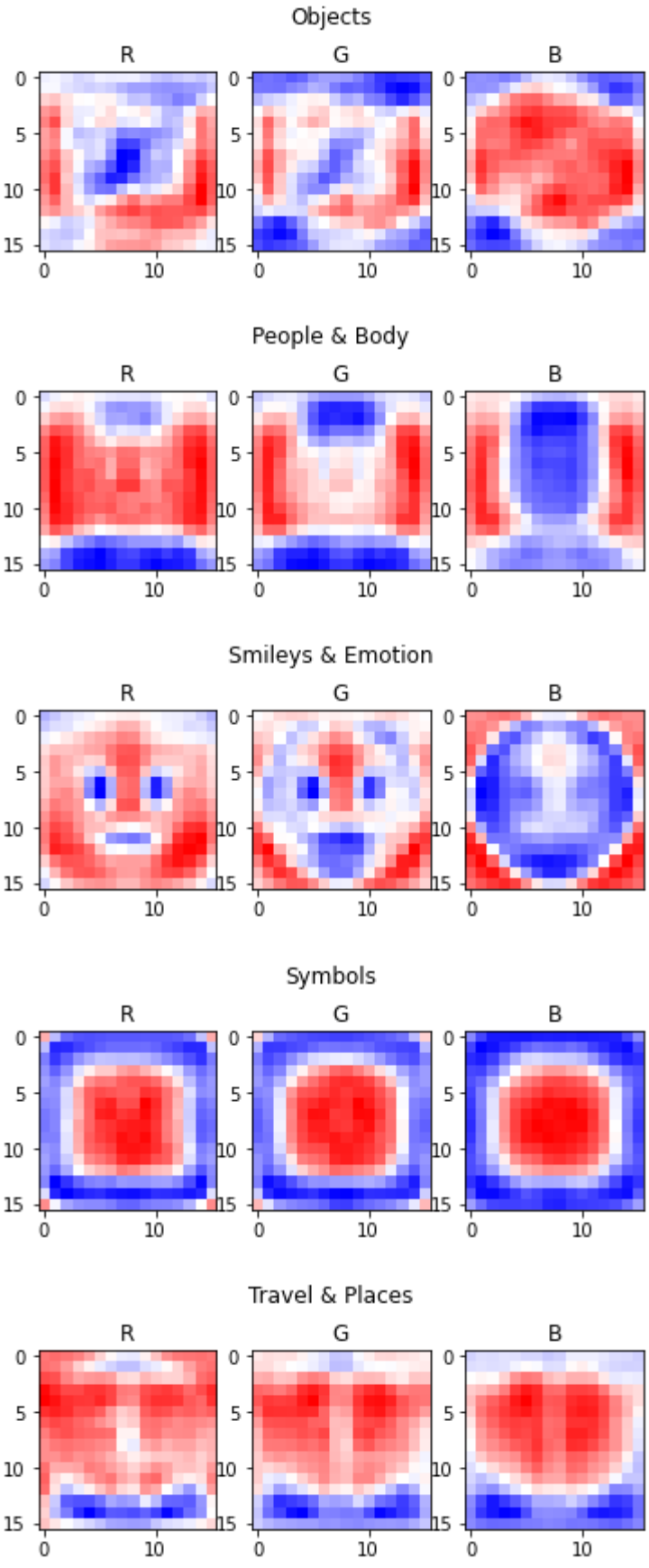
Hint: Note that the i^{th} set of weights may not match the order of categories. Use the *classes_* attribute of *LogisticRegression* to recover the category index.

In [251]:
```
def plot_logistic_weights(estimator):
    """
    Plots heatmaps showing the weights of the LogisticRegression estimator,
    with a separate plot for each class and for each colour channel.
    """
    assert isinstance(estimator, sklearn.linear_model.LogisticRegression)
    # Your implementation here here. Aim for 9-12 lines.
    estimator_coef = estimator.coef_
    estimator_class = estimator.classes_
    rgbTitle = ['R','G','B']
    counter = 0
    for f in estimator_coef:
        feature_importances = f.reshape(16,16,3)
        plt.figure(figsize = (6,2.5))
        plt.suptitle(estimator_class[counter])
        counter +=1
        for j in range(0,3):
            plt.subplot(1,3,j+1)
            plt.imshow(feature_importances[:,:,j], cmap = 'bwr')
            plt.title(rgbTitle[j])
```

Check your code by calling your function to plot the weights of the *LogisticRegression* classifier having *strongest* regularization (the one with *C*=1e-5).

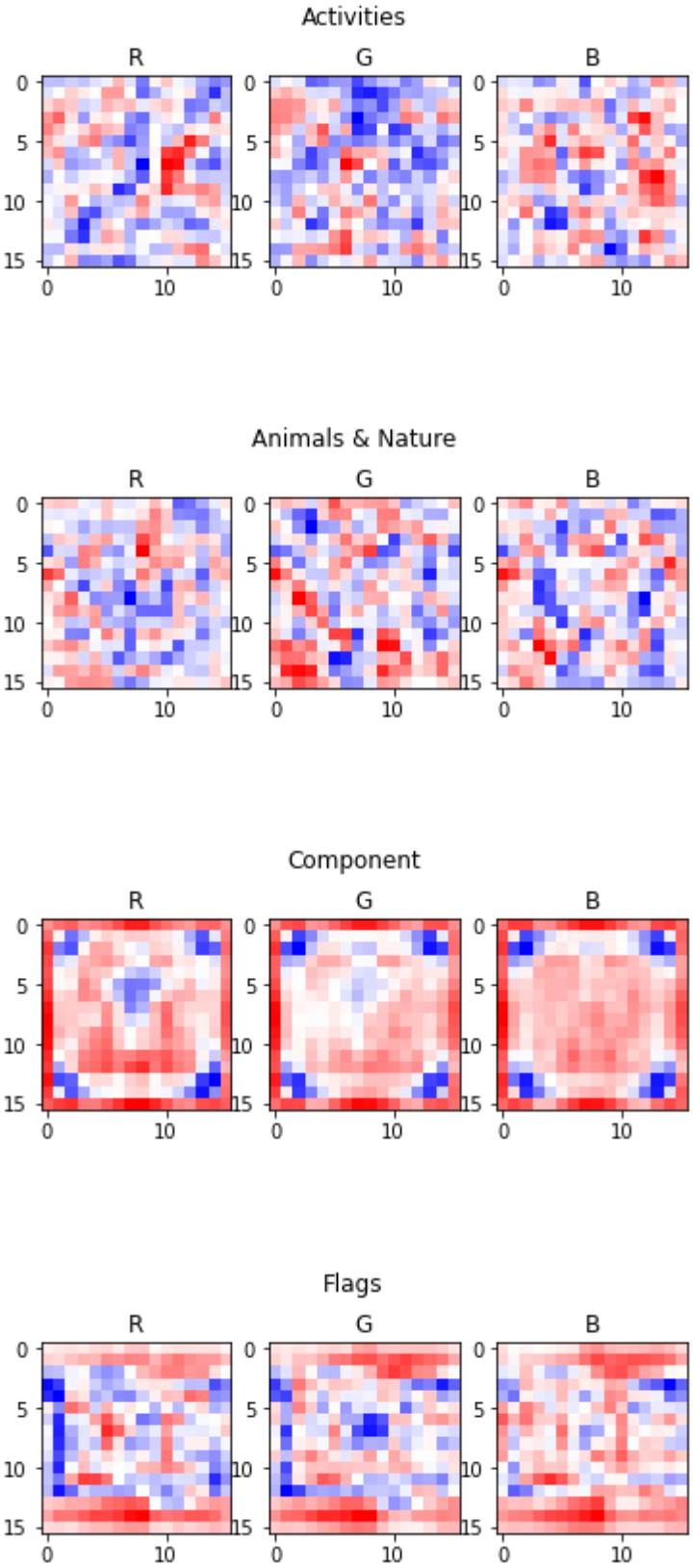
In [252]:
```
# Your code here (1 line)
plot_logistic_weights(log_estimators[0])
```

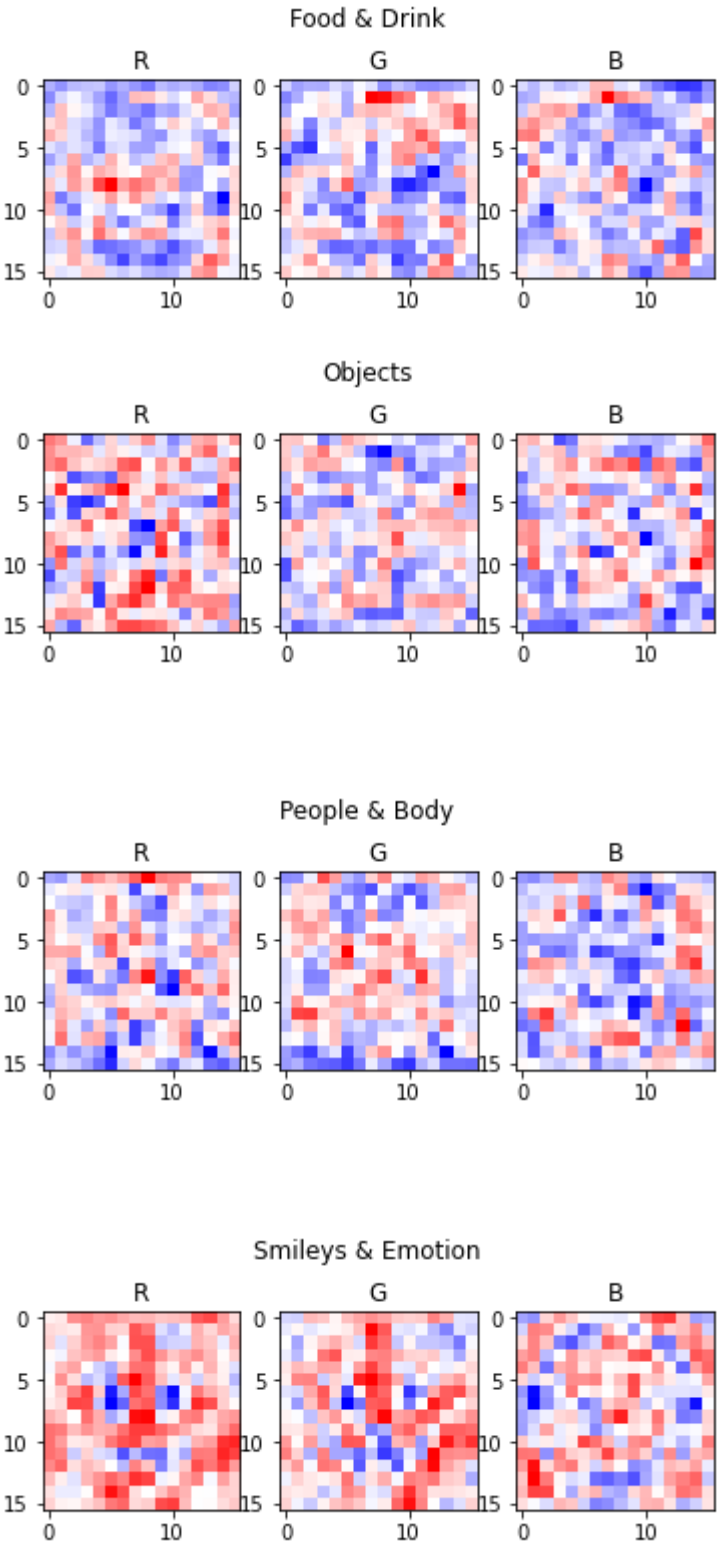


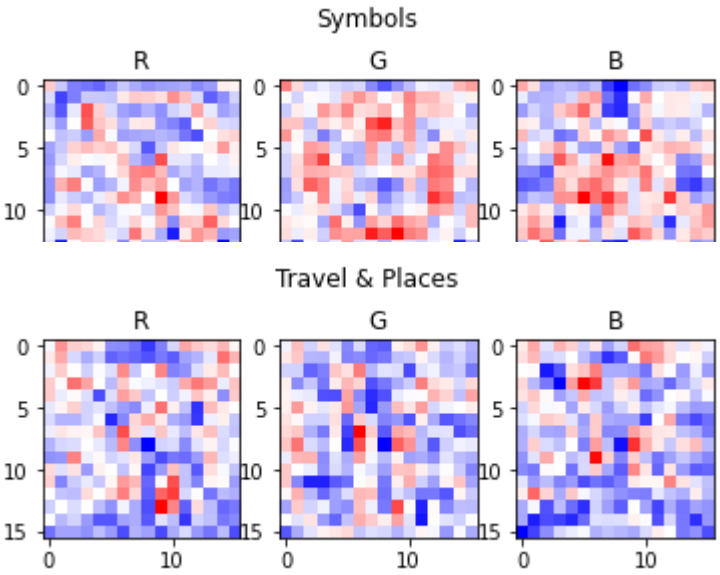


Plot the weights of your 'best' *LogisticRegression* instance.

In [255]:
```
# Your code here (1 line)
plot_logistic_weights(log_estimators[4])
```




Food & Drink
R
G
B
Objects
R
G
B
People & Body
R
G
B
Smileys & Emotion
R
G
B



Question. Was your best-performing *LogisiticRegression* classifier also the most interpretable? YES/NO then explain in 2-3 sentences.

```
Your answer here
No, it was not the most interpretable. Logistic regression model with C=1e-5
is far more interpretable than the best-performing one. There is no direct
proportion of C: inverse of regularization strength that is used to reduce
overfitting and the interpretability. The machine's interpretability is far
different from a human's
```

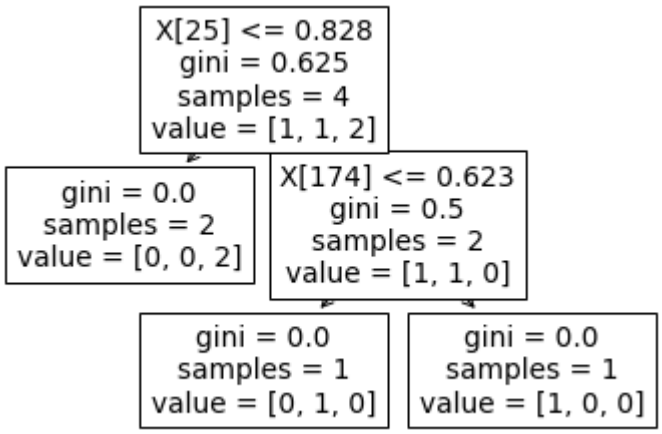
Q6 — Generating Python code for a tree [5 marks BONUS]

The goal here is to develop a deeper understanding of scikit-learn's decision tree and data structure, by generating an equivalent Python program, compiling it, and executing it. Most of the question is just learning how things work, and the part you have to do is at the very end.

Run the code cell below, replacing *X_trn* and *y_trn* with whatever you named your training set variables.

In [256]:
```
# Train a tiny tree on a tiny training set of only 4 instances
X_tiny = X_trn[:4]
y_tiny = y_trn[:4]
tiny_tree = sklearn.tree.DecisionTreeClassifier(max_depth=2, random_state=0).fit(

# Plot the tree and print the true and predicted labels
sklearn.tree.plot_tree(tiny_tree)
print("true:", y_tiny)
print("pred:", tiny_tree.predict(X_tiny))
```

```
true: ['Symbols' 'Flags' 'People & Body' 'Symbols']
pred: ['Symbols' 'Flags' 'People & Body' 'Symbols']
```



The above is meant as a simple easy-to-understand tree, much simpler than the real ones you trained in **Q3**.

Run the code cell below to see a function that traverses a *DecisionTreeClassifier*'s internal tree data structure and prints a message each time it encounters a split node or a leaf node. Notice how it corresponds to the plotted tree above.

In [262]:
```
def print_tree(tree):
    """Prints the structure of a DecisionTreeClassifier."""
    assert isinstance(tree, sklearn.tree.DecisionTreeClassifier)

    # Shorthand for some tree attributes
    left = tree.tree_.children_left     # left[i]: index of left  node when i is
    right = tree.tree_.children_right   # right[i]: index of right node when i is
    feature = tree.tree_.feature        # feature[i]: index feature to test when
    threshold = tree.tree_.threshold    # threshold[i]: threshold to use when i i
    votes = tree.tree_.value            # votes[i,j]: number of training examples
                                        #             node i while having class i

    def visit_subtree(i, depth):
        indent = "  "*depth
        if left[i] != right[i]:
            print("%snode %d: split on x[%d] <= %f" % (indent, i, feature[i], thr
            visit_subtree(left[i],  depth+1)
            visit_subtree(right[i], depth+1)
        else:
            label = tree.classes_[np.argmax(votes[i])]
            print("%snode %d: leaf label %s" % (indent, i, label))

    visit_subtree(0, 0)

print_tree(tiny_tree)
```

```
node 0: split on x[25] <= 0.827543
  node 1: leaf label Symbols
  node 2: split on x[174] <= 0.622561
    node 3: leaf label People & Body
    node 4: leaf label Flags
```

Call *print_tree* on your best *DecisionTreeClassifier* from **Q3**. (The output will be very long.)

In [263]:
```
# Your code here (1 line)
print_tree(tree_estimators[3])
```

```
                    node 36: split on x[528] <= 0.745384
                      node 37: split on x[55] <= 0.461936
                        node 38: leaf label Activities
                        node 39: leaf label People & Body
                      node 40: leaf label Objects
                  node 41: split on x[649] <= 0.749213
                    node 42: leaf label Symbols
                    node 43: split on x[688] <= 0.702469
                      node 44: leaf label Food & Drink
                      node 45: split on x[497] <= 0.524843
                        node 46: leaf label Activities
                        node 47: leaf label Objects
                node 48: split on x[665] <= 0.584400
                  node 49: split on x[125] <= 0.498393
                    node 50: split on x[88] <= 0.569947
                      node 51: split on x[443] <= 0.256584
                        node 52: leaf label Travel & Places
                        node 53: leaf label Symbols
                      node 54: split on x[101] <= 0.536942
                        node 55: split on x[400] <= 0.385855
```

Implement the *tree_to_code* function below. The idea is to transform a *DecisionTreeClassifier* instance into equivalent Python code, where the code is built up as a string. You may copy whatever code you want from *print_tree* as a starting point, but you must **add a comment for each new line of code**.

For example, if you called it with the *tiny_tree* from earlier, it might produce a string like below (although not necessarily an identical program, depending on the training set for *tiny_tree*).

```
>>> print(tree_to_code(tiny_tree))
def predict(x):
  if x[395] <= -0.642489:
    if x[174] <= -0.988780:
      return 4
    else:
      return 7
  else:
    return 0
```

In [388]:
```
def tree_to_code(tree):
    """
    Given a *DecisionTreeClassifier*, returns a string that
    defines a Python function that corresponds to how the
    decision tree makes predictions. The first line of the
    string is:

         "def predict(x):\n..."

    followed by lines of code for the logic of the tree.
    """
    # Your implementation here.
    """Prints the structure of a DecisionTreeClassifier."""
    assert isinstance(tree, sklearn.tree.DecisionTreeClassifier)

    # Shorthand for some tree attributes
    left = tree.tree_.children_left     # left[i]: index of left  node when i is
    right = tree.tree_.children_right   # right[i]: index of right node when i is
    feature = tree.tree_.feature        # feature[i]: index feature to test when
    threshold = tree.tree_.threshold    # threshold[i]: threshold to use when i i
    votes = tree.tree_.value            # votes[i,j]: number of training examples
                                        #             node i while having class i
    print('def predict(x):')
    counter = 0
    def visit_subtree(i, depth):
        if counter == 0:
            indent = "   "
        else:
            indent = "   "*depth
        if left[i] != right[i]:
            print("%sif x[%d] <= %f" % (indent,  feature[i], threshold[i]))
            print("%sreturn %d" %(indent+'   ',left[i]))
            visit_subtree(left[i],  depth+1)
            visit_subtree(right[i], depth+1)
        else:
            label = tree.classes_[np.argmax(votes[i])]
            print("%selse:" % (indent[0:-3]))
            print("%sreturn %d" %(indent+' ',right[i]))
    counter += 1
    visit_subtree(0, 0)
    return ''
```

Run the code cell below to check your implementation. You should see a program equivalent to the tiny decision tree you plotted earlier.

In [389]:
```
print(tree_to_code(tiny_tree))
```

```
def predict(x):
if x[25] <= 0.827543
   return 1
else:
    return -1
   if x[174] <= 0.622561
      return 3
   else:
       return -1
   else:
       return -1
```

Run the code cell below to define a utility function called *compile_func*. What this function does is it takes a string containing a single Python function definition, and compiles it, returning a *function* object that can be called to execute the code represented by the string.

In [390]:
```
def compile_func(python_code):
    """
    Compiles a string defining a Python function, and returns
    a reference to the callable function object that results.
    """
    symbols = {}                                # Dictionary to collect symbols that
    exec(python_code, None, symbols)            # Execute the string as if it were c
    assert len(symbols) == 1, "Expected python_code to define a function"
    function = next(iter(symbols.values()))  # Get reference to the object that w
    assert callable(function), "Expected python_code to define a function"
    return function
```

Run the code cell below to see a demo of how *compile_func* works.

In [391]:
```
example_code = """
def square(x):
    return x**2
"""

example_func = compile_func(example_code)
print(example_func)

for i in range(5):
    print(example_func(i))
```

```
<function square at 0x0000022B4521DEE0>
0
1
4
9
16
```

Write code to (a) convert your best *DecisionTreeClassifier* to a compiled Python function and (b)

assert that the compiled Python function produces the same predictions on the training set $\boldsymbol{X}_{\text{trn}}$. (Note that your Python function expects a 1-dimensional $\boldsymbol{x}$, whereas *DecisionTreeClassifier* expects a 2-dimensional $\boldsymbol{X}$.)

In [ ]:
```
# Your answer here. Aim for 2-4 lines.
```

Write code to compare the prediction speed of your *DecisionTreeClassifer* instance versus your pure-Python function. Specifically, you should use `%%time` to report the amount of time that *DecisionTreeClassifer* takes to generate predictions on the training set $\boldsymbol{X}_{\text{trn}}$.

In [ ]:
```
%%time
# Your code for timing your best DecisionTreeClassifier here. Aim for 1 line.
```

In [ ]:
```
%%time
# Your code for timing the Python version here. Aim for 1-2 lines.
```

Question. Which version was faster and by how much? What do you think explains this?

```
Your answer here
```