- **FA16-BL-INFO-I526-34917**
- **Applied Machine Learning**
- **Gerald Manipon, MS Data Science student - Indiana University**
- **gmanipon@iu.edu**
- **this jupyter notebook can be found here: https://github.com/pymonger/tropicalstorm-ml-analysis**

Will a tropical storm make landfall?

As a native of the Hawaiian Islands, the recent tropical storms in 2016 (Madeline and Lester) that threatened the state has sparked a very interesting and personal question. Can we predict whether or not a tropical storm will make landfall and possibly affect the safety and lives of the inhabitants? Currently there are real-time storm tracking models that enable our emergency response agencies to be as responsive as possible and to give up-to-the-minute status on potential threats. However, what does historical data say and can we predict landfall using machine learning on the initial observation of a tropical storm?

Generate the dataset

Dataset description

The source dataset I will use comes from the IBTrACS (International Best Track Archive for Climate Stewardship) project: https://www.ncdc.noaa.gov/ibtracs/index.php. This project is endorsed by the WMO (World Meteorological Organization) as an "official archiving and distribution resource for tropical cyclone best track data". The IBTrACS project provides datasets that:

- Contains the most complete global set of historical tropical cyclones available
- Combines information from numerous tropical cyclone datasets
- Simplifies inter-agency comparisons by providing storm data from multiple sources in one place
- Provides data in popular formats to facilitate analysis
- Checks the quality of storm inventories, positions, pressures, and wind speeds, passing the information on to the user

I will be using the IBTrACS-WMO NetCDF file that contains all storms: https://www.ncdc.noaa.gov/ibtracs/index.php?name=wmo-data. Since NetCDF is a self-describing format, info about the variables contained in this dataset can be introspected. Additional info about the variables is located here: ftp://eclipse.ncdc.noaa.gov/pub/ibtracs/v03r08/wmo/netcdf/README.netcdf.

I will be performing some ETL (extraction, transformation and loading) tasks to prepare and filter (remove records with missing values) the source dataset to a derived dataset which I will use for this analysis. The source dataset essentially aggregates every recorded tropical storm from different source agencies and provides time-series information of pertinent variables describing the storm as it progressed through its track. *Since I'm only interested in being able to predict whether or not the storm will make landfall based on the storm's genesis and initial observation, my derived dataset will be composed of features that are essentially the values of the source dataset variables at observation* t_0.

The class variable (prediction variable) will be derived from the source dataset's **landfall** variable:

- **landfall** { True, False }

```
short landfall(storm, time) ;
    landfall:long_name = "Minimum distance to land until next report (0=landfall)" ;
    landfall:units = "km" ;
    landfall:_FillValue = -999s ;
```

I will aggregate this variable into a single value of **True** or **False**. **True** signifies that the storm eventually made landfall at some point in the storm's track and **False** otherwise.

The features I will include from the source dataset are:

- **genesis_basin** { 0 = NA - North Atlantic, 1 = SA - South Atlantic, 2 = WP - West Pacific, 3 = EP - East Pacific, 4 = SP - South Pacific, 5 = NI - North Indian, 6 = SI - South Indian }

```
byte genesis_basin(storm) ;
    genesis_basin:long_name = "Basin of genesis" ;
    genesis_basin:units = " " ;
    genesis_basin:key = "0 = NA - North Atlantic\n",
"1 = SA - South Atlantic\n",
"2 = WP - West Pacific\n",
"3 = EP - East Pacific\n",
"4 = SP - South Pacific\n",
"5 = NI - North Indian\n",
"6 = SI - South Indian\n",
"7 = AS - Arabian Sea\n",
"8 = BB - Bay of Bengal\n",
"9 = EA - Eastern Australia\n",
"10 = WA - Western Australia\n",
"11 = CP - Central Pacific\n",
"12 = CS - Carribbean Sea\n",
"13 = GM - Gulf of Mexico\n",
"14 = MM - Missing" ;
    genesis_basin:Note = "Based on where the storm began" ;
```

  - the additional variable info at ftp://eclipse.ncdc.noaa.gov/pub/ibtracs/v03r08/wmo/netcdf/README.netcdf states that only values 0-6 are used for this variable thus I will be discretizing the values for this feature
- **sub_basin** of first observation { 0 = NA - North Atlantic, 1 = SA - South Atlantic, 2 = WP - West Pacific, 3 = EP - East Pacific, 4 = SP - South Pacific, 5 = NI - North Indian, 6 = SI - South Indian, 7 = AS - Arabian Sea, 8 = BB - Bay of Bengal, 9 = EA - Eastern Australia, 10 = WA - Western Australia, 11 = CP - Central Pacific, 12 = CS - Carribbean Sea, 13 = GM - Gulf of Mexico, 14 = MM - Missing }

```
byte sub_basin(storm, time) ;
    sub_basin:long_name = "Sub-Basin" ;
    sub_basin:units = " " ;
    sub_basin:key = "0 = NA - North Atlantic\n",
"1 = SA - South Atlantic\n",
"2 = WP - West Pacific\n",
"3 = EP - East Pacific\n",
"4 = SP - South Pacific\n",
"5 = NI - North Indian\n",
"6 = SI - South Indian\n",
"7 = AS - Arabian Sea\n",
"8 = BB - Bay of Bengal\n",
"9 = EA - Eastern Australia\n",
```

```
"10 = WA - Western Australia\n",
"11 = CP - Central Pacific\n",
"12 = CS - Carribbean Sea\n",
"13 = GM - Gulf of Mexico\n",
"14 = MM - Missing" ;
    sub_basin:Note = "Based on present location" ;
    sub_basin:_FillValue = '\201' ;
```

- this feature will remain numeric since there are 14 values

- **time** of first observation (MJD value) (REAL)

```
double time_wmo(storm, time) ;
    time_wmo:long_name = "Modified Julian Day" ;
    time_wmo:units = "days since 1858-11-17 00:00:00" ;
    time_wmo:_FillValue = 9.969209999999999e+36 ;
```

- **lon** (longitude) of first observation (REAL)

```
short lon_wmo(storm, time) ;
    lon_wmo:long_name = "Storm center longitude" ;
    lon_wmo:units = "degrees_east" ;
    lon_wmo:scale_factor = 0.0099999998f ;
    lon_wmo:_FillValue = -32767s ;
```

- **lat** (latitude) of first observation (REAL)

```
short lat_wmo(storm, time) ;
    lat_wmo:long_name = "Storm center latitude" ;
    lat_wmo:units = "degrees_north" ;
    lat_wmo:scale_factor = 0.0099999998f ;
    lat_wmo:_FillValue = -32767s ;
```

- **dist2land** (distance to land) of first observation (REAL)

```
short dist2land(storm, time) ;
    dist2land:long_name = "Distance to land" ;
    dist2land:units = "km" ;
    dist2land:_FillValue = -999s ;
```

- **msw** (maximum sustained wind) of first observation (REAL)

```
short pres_wmo(storm, time) ;
    pres_wmo:long_name = "Minimum Central Pressure (MCP)" ;
    pres_wmo:units = "mb" ;
    pres_wmo:scale_factor = 0.1f ;
    pres_wmo:_FillValue = -32767s ;
```

- **mcp** (minimum central pressure) of first observation (REAL)

```
short wind_wmo(storm, time) ;
    wind_wmo:long_name = "Maximum Sustained Wind (MSW)" ;
    wind_wmo:units = "kt" ;
    wind_wmo:scale_factor = 0.1f ;
    wind_wmo:_FillValue = -32767s ;
```

- **nature** (storm nature) { 0 = TS - Tropical, 1 = SS - Subtropical, 2 = ET - Extratropical, 3 = DS - Disturbance, 4 = MX - Mix of conflicting reports, 5 = NR - Not Reported, 6 = MM - Missing, 7 = - Missing }

```
    nature_wmo:long_name = "Storm nature" ;
    nature_wmo:key = "0 = TS - Tropical\n",
"1 = SS - Subtropical\n",
"2 = ET - Extratropical\n",
"3 = DS - Disturbance\n",
"4 = MX - Mix of conflicting reports\n",
"5 = NR - Not Reported\n",
"6 = MM - Missing\n",
"7 =  - Missing" ;
    nature_wmo:Note = "Based on classification from original sources" ;
    nature_wmo:_FillValue = '\201' ;
```

- this feature will be discretized

- **track_type** { 0 = main - cylclogenesis to cyclolysis, 1 = merge - cyclogenesis to merger, 2 = split - split to cyclolysis, 3 = other - split to merger }

```
byte track_type(storm) ;
    track_type:long_name = "Track type" ;
    track_type:key = "0 = main - cyclogenesis to cyclolysis\n",
"1 = merge - cyclogenesis to merger\n",
"2 = split - split to cyclolysis\n",
"3 = other - split to merger" ;
```

- this feature will be discretized

Open the source dataset

In [1]:

```
import os, sys, re, json, arff, time
from copy import deepcopy
from subprocess import check_output
import netCDF4 as NC
import numpy as np
import pandas as pd
from ipyleaflet import Map, GeoJSON
from astropy.time import Time
from IPython.display import display, HTML, Markdown


# get netcdf dataset
file = "Allstorms.ibtracs_wmo.v03r08.nc"
ds = NC.Dataset(file)
```

Extract, transform and load features (ETL)

In [2]:

```
# define dict for discrete features
disc_map = {
    "basin": {
          0: "NA", # North Atlantic
          1: "SA", # South Atlantic
          2: "WP", # West Pacific
          3: "EP", # East Pacific
          4: "SP", # South Pacific
          5: "NI", # North Indian
          6: "SI", # South Indian
          7: "AS", # Arabian Sea
          8: "BB", # Bay of Bengal
          9: "EA", # Eastern Australia
         10: "WA", # Western Australia
         11: "CP", # Central Pacific
         12: "CS", # Carribbean Sea
         13: "GM", # Gulf of Mexico
         14: "MM", # Missing
    },
    "nature": {
        0: "TS",  # Tropical
        1: "SS",  # Subtropical
        2: "ET",  # Extratropical
        3: "DS",  # Disturbance
        4: "MX",  # Mix of conflicting reports
        5: "NR",  # Not Reported
        6: "MM",  # Missing
        7: "MM2", # Also Missing
    },
    "track_type": {
        0: "main",  # cyclogenesis to cyclolysis
        1: "merge", # cyclogenesis to merger
        2: "split", # split to cyclolysis
        3: "other", # split to merger
    },
    "month": {
        1: "Jan",
        2: "Feb",
        3: "Mar",
        4: "Apr",
        5: "May",
        6: "Jun",
        7: "Jul",
        8: "Aug",
        9: "Sep",
       10: "Oct",
       11: "Nov",
       12: "Dec",
    }
}

# extract features from each hurricane and save into a list of dicts
data = []
landfall_count = 0

# compile regular expression for matching unnamed storms
unnamed_re = re.compile(r'(UNNAMED|NOT NAMED)')

for i in range(ds.dimensions['storm'].size):

    # get number of observations
    obs = ds.variables['numObs'][i]
    if obs <= 2: continue # skip if there are 2 or less observations

    # get storm id (storm names can be re-used so we need to track them uniquely)
    id = np.array_str(NC.chartostring(ds.variables['storm_sn'][i,:]))[2:-1]

    # extract filterable features first
    name = np.array_str(NC.chartostring(ds.variables['name'][i,:]))[2:-1]
    genesis_basin = ds.variables['genesis_basin'][i]
    sub_basin = ds.variables['sub_basin'][i,:obs-1]
    nature = ds.variables['nature_wmo'][i:,obs-1]

    # skip records that have missing values in features
    if genesis_basin == 14:
        continue
    # skipping this filter; this filters out the east pacific storms
    #if sub_basin[0] == 14:
    #    continue
    if nature[0] in (4, 5, 6, 7):
        continue

    # skip records with unnamed storms
    #if unnamed_re.search(name): continue

    # extract the rest of the features
    time_wmo = ds.variables['time_wmo'][i,:obs-1]
    time_iso = Time(time_wmo, format='mjd', scale='utc')

    # including the time feature as-is (absolute value) from the source doesn't make
    # sense for prediction; a better feature to derive from the time feature is the
    # month of year since this can give the algorithm insight into seasonal effects
    month = time_iso[0].datetime.month
```

```python
    # extract lon and handle wrapping issue
    lon = ds.variables['lon_wmo'][i,:obs-1]
    lon_diff = lon[0] - lon[-1]
    if lon_diff > 180.:
        lon[np.where(lon > 0)] -= 360.
    elif lon_diff < 180.:
        lon[np.where(lon < 0)] += 360.

    # extract other features
    lat = ds.variables['lat_wmo'][i,:obs-1]
    dist2land = ds.variables['dist2land'][i,:obs-1]
    msw = ds.variables['wind_wmo'][i,:obs-1]
    mcp = ds.variables['pres_wmo'][i,:obs-1]
    tt = ds.variables['track_type'][i]

    # extract the class feature: landfall; if at any time in the storm's track
    # it makes landfall, then the class feature landfall == True; otherwise it
    # will be landfall == False
    landfall = (ds.variables['landfall'][i,:obs-1] == 0).any()

    # create GeoJSON of storm track
    ls = {
        "type": "LineString",
        "coordinates": np.dstack((lon, lat))[0].tolist(),
    }

    # create feature for leaflet display;
    # stuff features into a message info for on_hover display
    msg = "{} {} {} {} {} {} {} {} {} {} {} {} {}".format(i, name, obs, genesis_basin, sub_basin[0],
                                                        time_iso[0].iso, lon[0], lat[0], dist2land[0],
                                                        msw[0], mcp[0], nature[0], tt, landfall)
    ls_feature = {
        "type": "Feature",
        "properties": { "msg": msg },
        "geometry": ls,
    }

    # create data dict
    data.append({
        "id": id,
        "name": name,
        "genesis_basin": disc_map['basin'][genesis_basin],
        "sub_basin": disc_map['basin'][sub_basin[0]],
        "time": time_wmo[0],
        "month": disc_map['month'][month],
        "lon": lon[0],
        "lat": lat[0],
        "dist2land": dist2land[0],
        "msw": msw[0],
        "mcp": mcp[0],
        "nature": disc_map['nature'][nature[0]],
        "track_type": disc_map['track_type'][tt],
        "landfall": landfall,
        "feature": json.dumps(ls_feature),
    })

    # tally landfall
    if landfall: landfall_count += 1

# create data frame
df = pd.DataFrame(data)

#

# print class label distribution of filtered source dataset
display(Markdown("### class label distribution of filtered source dataset"))
display(Markdown("* total storms: {}".format(len(data))))
display(Markdown("* total storms with class variable landfall == True: {}".format(landfall_count)))
display(Markdown("* total storms with class variable landfall == False: {}".format(len(data)-landfall_count)))
```

```
WARNING: ErfaWarning: ERFA function "d2dtf" yielded 1 of "dubious year (Note 5)" [astropy._erfa.core]
```

class label distribution of filtered source dataset

- total storms: 4836
- total storms with class variable landfall == True: 2315
- total storms with class variable landfall == False: 2521

In [3]:

```
display(df[['id', 'name', 'genesis_basin', 'sub_basin', 'time', 'month', 'lon', 'lat',
            'dist2land', 'msw', 'mcp', 'nature', 'track_type', 'landfall']])
```

	id	name	genesis_basin	sub_basin	time	month	lon	lat	dist2land	msw	mcp	nature	track_type	landfall
0	1851175N26270	UNNAMED	NA	GM	-2702.00	Jun	265.200012	28.000000	116	80.0	0.0	TS	main	True
1	1851228N13313	UNNAMED	NA	NA	-2650.00	Aug	312.000000	13.400000	1048	40.0	0.0	TS	main	True
2	1851256N33287	UNNAMED	NA	NA	-2622.00	Sep	286.500000	32.500000	370	50.0	0.0	TS	main	False
3	1851289N29282	UNNAMED	NA	NA	-2589.00	Oct	282.000000	28.699999	244	40.0	0.0	TS	main	True
4	1852232N21293	UNNAMED	NA	NA	-2281.00	Aug	292.899994	20.500000	234	60.0	0.0	TS	main	True
5	1852247N14309	UNNAMED	NA	GM	-2260.00	Sep	269.600006	26.400000	302	70.0	0.0	TS	main	True
6	1852249N17296	UNNAMED	NA	CS	-2264.00	Sep	295.899994	17.000000	208	70.0	0.0	TS	main	True
7	1852264N13309	UNNAMED	NA	NA	-2247.00	Sep	301.500000	16.100000	662	50.0	0.0	TS	main	False
8	1852278N14293	UNNAMED	NA	CS	-2233.00	Oct	286.200012	17.000000	111	90.0	0.0	TS	main	True
9	1853242N12336	UNNAMED	NA	NA	-1905.00	Aug	336.799988	12.099999	682	40.0	0.0	TS	main	False
10	1853251N37307	UNNAMED	NA	NA	-1896.00	Sep	307.000000	37.000000	1067	100.0	0.0	TS	merge	False
11	1853253N41303	UNNAMED	NA	NA	-1894.75	Sep	300.600006	39.899998	601	90.0	0.0	TS	merge	False
12	1853269N26298	UNNAMED	NA	NA	-1878.00	Sep	298.000000	25.799999	906	50.0	0.0	TS	main	False
13	1853291N32280	UNNAMED	NA	NA	-1855.00	Oct	281.500000	27.500000	167	70.0	0.0	TS	main	False
14	1854176N26268	UNNAMED	NA	GM	-1606.00	Jun	267.500000	26.000000	380	60.0	0.0	TS	main	True
15	1854246N25300	UNNAMED	NA	NA	-1532.00	Sep	283.399994	26.400000	201	110.0	0.0	TS	main	True
16	1854261N28266	UNNAMED	NA	GM	-1521.00	Sep	266.399994	28.199999	168	90.0	0.0	TS	main	True
17	1854293N25292	UNNAMED	NA	NA	-1489.00	Oct	292.399994	25.000000	652	50.0	0.0	TS	main	False
18	1855222N44318	UNNAMED	NA	NA	-1195.00	Aug	318.000000	44.000000	905	90.0	0.0	TS	main	False
19	1855236N12304	UNNAMED	NA	NA	-1181.00	Aug	304.100006	12.000000	562	50.0	0.0	TS	main	True
20	1855252N20274	UNNAMED	NA	GM	-1159.00	Sep	270.899994	26.699999	256	70.0	0.0	TS	main	True
21	1856221N25277	UNNAMED	NA	GM	-830.00	Aug	276.100006	25.000000	240	70.0	0.0	TS	main	True
22	1856226N11308	UNNAMED	NA	NA	-826.00	Aug	303.899994	12.099999	556	70.0	0.0	TS	main	False
23	1856232N33285	UNNAMED	NA	NA	-820.00	Aug	284.500000	32.500000	262	50.0	0.0	TS	main	True
24	1856235N13302	UNNAMED	NA	NA	-814.00	Aug	290.200012	21.000000	157	70.0	0.0	TS	main	True
25	1856262N32311	UNNAMED	NA	NA	-790.00	Sep	311.200012	32.000000	1667	50.0	0.0	TS	split	False
26	1857181N34286	UNNAMED	NA	NA	-505.00	Jun	285.500000	34.000000	192	50.0	0.0	TS	main	False
27	1857249N27286	UNNAMED	NA	NA	-436.25	Sep	287.000000	26.500000	521	40.0	0.0	ET	main	True
28	1857265N33287	UNNAMED	NA	NA	-421.00	Sep	286.500000	32.500000	370	70.0	0.0	TS	main	False
29	1857267N16305	UNNAMED	NA	NA	-419.00	Sep	305.299988	16.000000	903	50.0	0.0	TS	main	True
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
4806	2014212N11242	ISELLE	EP	MM	56868.50	Jul	242.100006	11.300000	1544	20.0	1010.0	TS	main	False
4807	2014214N13249	JULIO	EP	MM	56871.00	Aug	248.800003	12.500000	1015	25.0	1008.0	TS	main	False
4808	2014225N15255	KARINA	EP	MM	56881.50	Aug	255.000000	15.099999	392	25.0	1008.0	DS	main	False
4809	2014229N16246	LOWELL	EP	MM	56886.00	Aug	246.300003	16.000000	863	20.0	1008.0	DS	main	False
4810	2014234N12261	MARIE	EP	MM	56891.00	Aug	261.200012	12.300000	428	30.0	1007.0	DS	main	False
4811	2014236N22288	CRISTOBAL	NA	NA	56892.75	Aug	287.799988	21.500000	89	30.0	1005.0	ET	main	True
4812	2014245N19268	DOLLY	NA	GM	56901.50	Sep	267.700012	19.199999	67	25.0	1009.0	DS	main	True
4813	2014246N17254	NORBERT	EP	MM	56902.50	Sep	253.500000	17.000000	304	35.0	1004.0	DS	main	False
4814	2014248N19129	FENGSHEN	WP	MM	56905.50	Sep	127.599998	20.299999	595	0.0	1004.0	ET	main	False
4815	2014253N13244	SIXTEEN	EP	MM	56910.25	Sep	243.500000	12.500000	1351	25.0	1009.0	DS	main	False
4816	2014253N14260	ODILE	EP	MM	56909.50	Sep	259.899994	13.500000	352	25.0	1007.0	TS	main	True
4817	2014254N10142	KALMAEGI	WP	MM	56911.75	Sep	134.000000	13.500000	933	0.0	1004.0	TS	main	True
4818	2014254N14327	EDOUARD	NA	NA	56910.75	Sep	326.799988	13.700000	1696	25.0	1009.0	DS	main	False
4819	2014259N11262	POLO	EP	MM	56916.00	Sep	262.399994	11.400000	482	35.0	1007.0	DS	main	False
4820	2014260N13135	FUNG-WONG	WP	MM	56917.00	Sep	135.000000	12.599999	1017	0.0	1002.0	ET	main	True
4821	2014267N15257	RACHEL	EP	MM	56924.00	Sep	257.399994	14.500000	352	30.0	1006.0	DS	main	False
4822	2014267N18150	KAMMURI	WP	MM	56923.50	Sep	149.599991	17.900000	2126	0.0	1004.0	ET	main	False
4823	2014271N10160	PHANFONE	WP	MM	56928.25	Sep	157.099991	11.000000	1681	0.0	1004.0	ET	main	True
4824	2014274N15260	SIMON	EP	MM	56930.50	Sep	259.899994	14.700000	227	25.0	1008.0	DS	main	True
4825	2014275N06166	VONGFONG	WP	MM	56932.50	Oct	162.099991	7.300000	1625	0.0	1006.0	ET	main	True
4826	2014283N22299	FAY	NA	NA	56940.00	Oct	298.799988	22.000000	610	35.0	1007.0	TS	main	False
4827	2014284N10231	ANA	EP	MM	56940.50	Oct	231.300003	9.700000	2429	20.0	1009.0	ET	main	False
4828	2014285N16305	GONZALO	NA	NA	56941.75	Oct	305.100006	16.400000	917	25.0	1010.0	ET	main	False
4829	2014290N14261	TRUDY	EP	MM	56947.25	Oct	261.399994	13.900000	250	25.0	1007.0	TS	main	True
4830	2014294N20265	HANNA:INVEST	NA	GM	56951.00	Oct	264.700012	19.500000	88	25.0	1002.0	DS	main	True
4831	2014303N11261	VANCE	EP	MM	56959.75	Oct	260.899994	10.700000	609	30.0	1007.0	TS	main	False
4832	2014303N13141	NURI	WP	MM	56960.00	Oct	140.899994	12.599999	1599	0.0	1004.0	ET	main	False
4833	2014329N08131	SINLAKU	WP	MM	56987.00	Nov	128.000000	8.400000	173	0.0	1002.0	TS	main	True
4834	2014334N02156	HAGUPIT	WP	MM	56991.50	Nov	156.000000	2.600000	797	0.0	1006.0	TS	main	True
4835	2014362N07130	JANGMI	WP	MM	57019.00	Dec	128.699997	7.400000	231	0.0	1006.0	TS	main	True

4836 rows × 14 columns

Generate random sampling of each prediction (class) value and generate ARFF file

In this step, we sample 500 records where the class label "landfall" is True and 500 records where it is False. The merger of these 2 samples will comprise the training set used in this analysis. The rest of the records will comprise the test set. We also serialize the training and test datasets to ARFF files (for use in Weka) and HDF5 for use with scikit-learn.

The final feature set that we include in our input files is:

- nature
- track_type
- month
- lon
- lat
- dist2land
- msw
- mcp
- landfall (class)

We filtered out the genesis_basin feature because it is a categorical feature that is based on the geographic location of the initial storm observation. In our derived dataset, we have the storm track's latitude and longitude values as features of type float to provide geographic input to the learners.

We also filtered out the time feature because it is a measure of the storm's temporal location on an absolute timescale. Since our main goal is for the prediction of landfall of future storms, this feature in its original form does not provide any value to the learners. However, we can extract valuable temporal information from this feature by extracting the month of year for the storm track. By providing the month of year to the learning algorithms, we might possibly provide a valuable and insightful dimension to the dataset as it pertains to seasonal effects and trends.

In [4]:

```
# randomly sample 500 records of each class value (these will be the training set)
landfall_sample = df.loc[df['landfall'] == True].sample(500)
nolandfall_sample = df.loc[df['landfall'] == False].sample(500)

# get the rest of the records to be used as the test set
test_set = df[-df['id'].isin(landfall_sample['id']) &
              -df['id'].isin(nolandfall_sample['id'])]

# base ARFF dict for create ARFF files
arff_data = {
    "relation": "tropicalstorms",
    "description": "IBTrACS (International Best Track Archive for Climate Stewardship) tropical storm database",
    "attributes": [
        #("genesis_basin", ["NA", "SA", "WP", "EP", "SP", "NI", "SI"]),
        ("nature", ["TS", "SS", "ET", "DS"]),
        ("track_type", ["main", "merge", "split", "other"]),
        ("month", ["Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"]),
        #("time", "DATE"),
        #("time", "REAL"),
        ("lon", "REAL"),
        ("lat", "REAL"),
        ("dist2land", "REAL"),
        ("msw", "REAL"),
        ("mcp", "REAL"),
        ("landfall", ["True", "False"]),
    ],
    "data": [],
}

# create ARFF file for training set
arff_data_training = deepcopy(arff_data)
for r in landfall_sample.itertuples():
    arff_data_training['data'].append([r.nature, r.track_type, r.month,
                                       r.lon, r.lat, r.dist2land, r.msw,
                                       r.mcp, r.landfall])
for r in nolandfall_sample.itertuples():
    arff_data_training['data'].append([r.nature, r.track_type, r.month,
                                       r.lon, r.lat, r.dist2land, r.msw,
                                       r.mcp,  r.landfall])
with open('tropicalstorms-trainingset.arff', 'w') as f:
    arff.dump(arff_data_training, f)

# create ARFF file for test set
arff_data_test = deepcopy(arff_data)
for r in test_set.itertuples():
    arff_data_test['data'].append([r.nature, r.track_type, r.month,
                                   r.lon, r.lat, r.dist2land, r.msw,
                                   r.mcp, r.landfall])
with open('tropicalstorms-testset.arff', 'w') as f:
    arff.dump(arff_data_test, f)

# print breakdown of records used for training and test sets
display(Markdown("### breakdown of dataset records used for training and test sets"))
display(Markdown("* total storms: {}".format(len(df))))
display(Markdown("* total storms used for training set : {}".format(len(landfall_sample) + len(nolandfall_sample))))
display(Markdown("* total storms used for test set: {}".format(len(test_set))))

# print class label distribution of randomly sampled datasets of each class value
display(Markdown("### training set class label distribution of randomly sampled datasets of each class value"))
display(Markdown("* total training set storms: {}".format(len(landfall_sample) + len(nolandfall_sample))))
display(Markdown("* total training set storms with class variable landfall == True : {}".format(len(landfall_sample))))
display(Markdown("* total training set storms with class variable landfall == False: {}".format(len(nolandfall_sample))))
display(Markdown("* total test set storms: {}".format(len(test_set))))
```

breakdown of dataset records used for training and test sets

- total storms: 4836
- total storms used for training set : 1000
- total storms used for test set: 3836

training set class label distribution of randomly sampled datasets of each class value

- total training set storms: 1000
- total training set storms with class variable landfall == True : 500
- total training set storms with class variable landfall == False: 500
- total test set storms: 3836

Save DataFrame datasets (all, landfall/nolandfall training sets and test set) as tables in HDF5

In [5]:

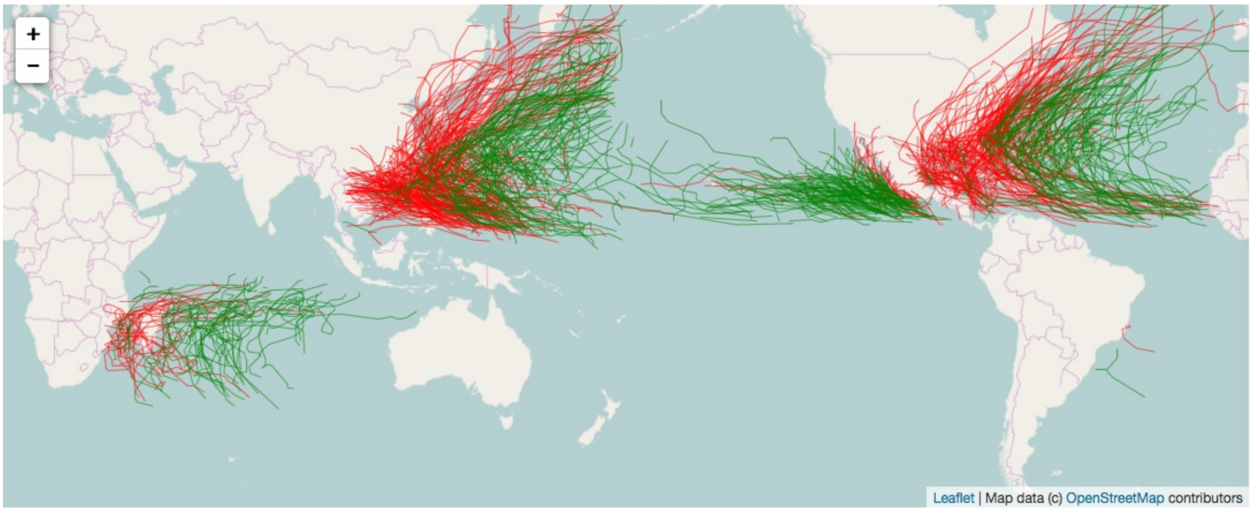
```
# save
df.to_hdf("tropicalstorms-all.h5", "tropicalstorms", format="table", complib="zlib", complevel=9)
landfall_sample.to_hdf("tropicalstorms-trainingset-landfall.h5", "tropicalstorms", format="table", complib="zlib", complevel=9)
nolandfall_sample.to_hdf("tropicalstorms-trainingset-nolandfall.h5", "tropicalstorms", format="table", complib="zlib", complevel=9)
test_set.to_hdf("tropicalstorms-testset.h5", "tropicalstorms", format="table", complib="zlib", complevel=9)
```

Visualize the dataset storms tracks

In [6]:

```
# set styles
trainingset_landfall_style = {
    "color": "red",
    "weight": 1,
}
trainingset_nolandfall_style = {
    "color": "green",
    "weight": 1,
}
testset_style = {
    "color": "yellow",
    "weight": 1,
}
hover_style = {
    "weight": 5,
}

# hover handler
def hover_handler(event=None, id=None, properties=None):
    sys.stdout.write("\r" + properties['msg'])
    sys.stdout.flush()

# show map
m = Map(center=[0, 180], zoom=2)
m
```

```
6075 KALUNDE 57 6 14 2003-03-03 06:00:00.000 80.4000015258789 -10.399999618530273 1826 20.0 1005.0 2 0
```



In [7]:

```
# add training set storm tracks that made landfall
for r in landfall_sample.itertuples():
    l = GeoJSON(data=json.loads(r.feature), style=trainingset_landfall_style, hover_style=hover_style)
    l.on_hover(hover_handler)
    m.add_layer(l)
```

In [8]:

```
# add training set storm tracks that didn't make landfall
for r in nolandfall_sample.itertuples():
    l = GeoJSON(data=json.loads(r.feature), style=trainingset_nolandfall_style, hover_style=hover_style)
    l.on_hover(hover_handler)
    m.add_layer(l)
```

Machine learning analysis

In our ML analysis of the tropical storm dataset we derived from the IBTrACS dataset, we will use scikit-learn to develop models using 5 different learning algorithms:

- Decision Tree
- Naive Bayes
- Logistic Regression
- Nearest Neighbors
- Support Vector Machines (SVM)

We will assess the prediction performance and potential pitfalls of each of these algorithms as well as explore the different parameter settings that pertain to each algorithm and how they can be adjusted to improve performance without overfitting. We will employ ShuffleSplit cross-validation entirely on the training set to perform this hyperparameter setting exploration so that knowledge about the test set doesn't leak into our model. Only after we've settled on the hyperparameter settings will we run the prediction on the test set.

Additionally we will create rescaled copies of the training and test sets since many machine learning methods are more effective if the data attributes have the same scale. We will create a copy of the datasets where the data has been normalized and rescaled into the range of 0 and 1. We'll also create another copy where the data has been standardized and the distribution of each feature has been shifted to have a mean of 0 and a standard deviation of 1. In our analysis, we will assess how these data rescaling methods can help improve classifier performance by including them in our cross-validation studies. From here on, we will refer to the original unscaled dataset as **raw**, the normalized dataset as **normalized** or **norm**, and the standardized dataset as **standardized** or **std**.

Our scenario of being able to predict whether or not a storm will make landfall is similar to the scenario discussed in class whereby we need to identify individuals with a certain contagious and deadly disease (zika, hanta, bubonic plague). In these cases, we prioritize identifying the true positives and the minimization of false negatives. In other words, we want to make sure we choose the classifier that provides the best accuracy in regards to identifying storms that will hit land and minimizes those predictions that identify storms that won't not hit land but actually did in truth.

With that in mind, our analysis shall proceed with the caveat that the cost associated with false negatives is far greater than the cost associated with false positives. As such, we do not want to judge the learning algorithms entirely on accuracy but in conjunction with these cost associations. Thus we want to choose the algorithm that maximizes the true positive rate or sensitivity [TP/(TP+FN)] and essentially the AUC (area under the curve) value of the ROC (receiver operating characteristics) curve.

Before we proceed, let's extract the training and test datasets, create normalized and standardized copies of these datasets, and define some global variables and functions that will be used across the analyses of each learning algorithm.

In [9]:

```
import io, itertools
import matplotlib.pyplot as plt
from matplotlib.font_manager import FontProperties
import numpy as np
import pandas as pd
from IPython.display import display, HTML, Markdown
from sklearn import metrics, preprocessing, model_selection, utils
from scipy import interp
from itertools import cycle


# enable inline images
get_ipython().enable_matplotlib('inline')

# globals
FEATURES = ['nature', 'track_type', 'month', 'lon', 'lat', 'dist2land', 'msw', 'mcp']
LABEL = 'landfall'
CATEGORY_FEATURES = ['nature', 'track_type', 'month']
CLASS_LABELS = ['nolandfall', 'landfall'] # [0, 1]

# dataset file names
trainset_landfall_file = "tropicalstorms-trainingset-landfall.h5"
trainset_nolandfall_file = "tropicalstorms-trainingset-nolandfall.h5"
testset_file = "tropicalstorms-testset.h5"

# read in training set and test set
trainset_landfall = pd.read_hdf(trainset_landfall_file)
trainset_nolandfall = pd.read_hdf(trainset_nolandfall_file)
testset = utils.shuffle(pd.read_hdf(testset_file))

# join training sets and shuffle
trainset = utils.shuffle(trainset_landfall.append(trainset_nolandfall))


def extract_features(df):
    """Extract features."""
    return df[FEATURES]


def extract_label(df):
    """Extract class feature."""
    return df[LABEL].astype('int')


def get_dummies(df, categories):
    """Return data frame where categorical columns are replaced with
    dummy/indicator values."""
    return pd.get_dummies(df, prefix_sep='=', columns=categories)


def plot_confusion_matrix(cm, classes, normalize=False, title='Confusion matrix', cmap=plt.cm.Blues):
    """
    This function prints and plots the confusion matrix.
    Normalization can be applied by setting `normalize=True`.
    """
    plt.figure()
    plt.imshow(cm, interpolation='nearest', cmap=cmap)
    plt.title(title)
    plt.colorbar()
    tick_marks = np.arange(len(classes))
    plt.xticks(tick_marks, classes, rotation=45)
    plt.yticks(tick_marks, classes)

    if normalize:
        cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
        #print("Normalized confusion matrix")
    else:
        #print('Confusion matrix, without normalization')
        pass

    #print(cm)

    thresh = cm.max() / 2.
    for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
        plt.text(j, i, cm[i, j],
                 horizontalalignment="center",
                 color="white" if cm[i, j] > thresh else "black")

    plt.tight_layout()
    plt.ylabel('True label')
    plt.xlabel('Predicted label')

    data = io.BytesIO()
    plt.savefig(data)


def plot_roc(y_truth, y_scores):
    """This function prints the ROC (Receiver Operating Characteristics) plot."""

    # compute ROC curve and ROC area
    fpr, tpr, _ = metrics.roc_curve(y_truth, y_scores)
    roc_auc = metrics.auc(fpr, tpr)
    table = "| **metric** | **score** |\n"
    table += "| --- | --- |\n"
    table += "| AUC | %f |\n" % roc_auc
    display(Markdown("**ROC:**"))
    display(Markdown(table))

    # plot
    font_prop = FontProperties()
    font_prop.set_size('small')
    plt.figure()
    plt.plot(fpr, tpr, color='darkorange', label='ROC curve (area = %f)' % roc_auc)
```

```python
    plt.plot([0, 1], [0, 1], color='navy', linestyle='--')
    plt.xlim([0.0, 1.0])
    plt.ylim([0.0, 1.05])
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.title('ROC curve for landfall == true')
    plt.legend(loc="upper left", bbox_to_anchor=(1,1), prop=font_prop)

    data = io.BytesIO()
    plt.savefig(data)


def print_metrics(truth, pred):
    """This function prints prediction metrics."""
    accuracy_score = metrics.accuracy_score(truth, pred)
    average_precision_score = metrics.average_precision_score(truth, pred)
    f1_score = metrics.f1_score(truth, pred)
    recall_score = metrics.recall_score(truth, pred)
    table = "| **metric** | **score** |\n"
    table += "| --- | --- |\n"
    table += "| accuracy_score | %f |\n" % accuracy_score
    table += "| average_precision_score | %f |\n" % average_precision_score
    table += "| f1_score | %f |\n" % f1_score
    table += "| recall_score | %f |\n" % recall_score
    display(Markdown("**classification scores:**"))
    display(Markdown(table))
    display(Markdown("**classification report**:"))
    display(HTML("<pre>%s</pre>" % metrics.classification_report(truth, pred, target_names=CLASS_LABELS)))


def cross_validation_metrics(clf, trainset, folds=10):
    """Run ShuffleSplit cross-validation on the classifier, print metrics, and plot ROC."""

    # extract training set features
    X_train = get_dummies(extract_features(trainset), CATEGORY_FEATURES)

    # generate normalized training set
    X_train_norm = preprocessing.normalize(X_train)

    # generate standardized training set
    X_train_std = preprocessing.normalize(X_train)

    # extract labels
    y_train = extract_label(trainset)

    # shuffle and split
    ss = model_selection.ShuffleSplit(n_splits=folds, test_size=.25)

    # score accuracy on raw, normalized, and standardized
    scores = model_selection.cross_val_score(clf, X_train, y_train, cv=ss)
    scores_norm = model_selection.cross_val_score(clf, X_train_norm, y_train, cv=ss)
    scores_std = model_selection.cross_val_score(clf, X_train_std, y_train, cv=ss)

    # ROC plot
    font_prop = FontProperties()
    font_prop.set_size('xx-small')
    plt.figure()

    # plot ROC curves for raw dataset
    i = 0
    mean_tpr = 0.0
    mean_fpr = np.linspace(0, 1, 100)
    for train, test in ss.split(X_train):
        probas_ = clf.fit(X_train.take(train), y_train.take(train)).predict_proba(X_train.take(test))
        # Compute ROC curve and area under the curve
        fpr, tpr, thresholds = metrics.roc_curve(y_train.take(test), probas_[:, 1])
        mean_tpr += interp(mean_fpr, fpr, tpr)
        mean_tpr[0] = 0.0
        roc_auc = metrics.auc(fpr, tpr)
        plt.plot(fpr, tpr, color='red', label='raw ROC fold %d (area = %f)' % (i, roc_auc))
        i += 1
    mean_tpr /= i
    mean_tpr[-1] = 1.0
    mean_auc_raw = metrics.auc(mean_fpr, mean_tpr)

    # plot ROC curves for normalized dataset
    i = 0
    mean_tpr = 0.0
    mean_fpr = np.linspace(0, 1, 100)
    for train, test in ss.split(X_train_norm):
        probas_ = clf.fit(X_train.take(train), y_train.take(train)).predict_proba(X_train.take(test))
        # Compute ROC curve and area under the curve
        fpr, tpr, thresholds = metrics.roc_curve(y_train.take(test), probas_[:, 1])
        mean_tpr += interp(mean_fpr, fpr, tpr)
        mean_tpr[0] = 0.0
        roc_auc = metrics.auc(fpr, tpr)
        plt.plot(fpr, tpr, color='cyan', label='normalized ROC fold %d (area = %f)' % (i, roc_auc))
        i += 1
    mean_tpr /= i
    mean_tpr[-1] = 1.0
    mean_auc_norm = metrics.auc(mean_fpr, mean_tpr)

    # plot ROC curves for standardized dataset
    i = 0
    mean_tpr = 0.0
    mean_fpr = np.linspace(0, 1, 100)
    for train, test in ss.split(X_train_std):
        probas_ = clf.fit(X_train.take(train), y_train.take(train)).predict_proba(X_train.take(test))
        # Compute ROC curve and area under the curve
        fpr, tpr, thresholds = metrics.roc_curve(y_train.take(test), probas_[:, 1])
        mean_tpr += interp(mean_fpr, fpr, tpr)
        mean_tpr[0] = 0.0
        roc_auc = metrics.auc(fpr, tpr)
        plt.plot(fpr, tpr, color='green', label='standardized ROC fold %d (area = %f)' % (i, roc_auc))
        i += 1
    mean_tpr /= i
    mean_tpr[-1] = 1.0
    mean_auc_std = metrics.auc(mean_fpr, mean_tpr)

    plt.plot([0, 1], [0, 1], linestyle='--', color='k', label='Luck')
    plt.xlim([0.0, 1.0])
    plt.ylim([0.0, 1.05])
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.title('ROC curve for landfall == true')
    plt.legend(loc="upper left", bbox_to_anchor=(1,1), prop=font_prop)
    plt.show()
```

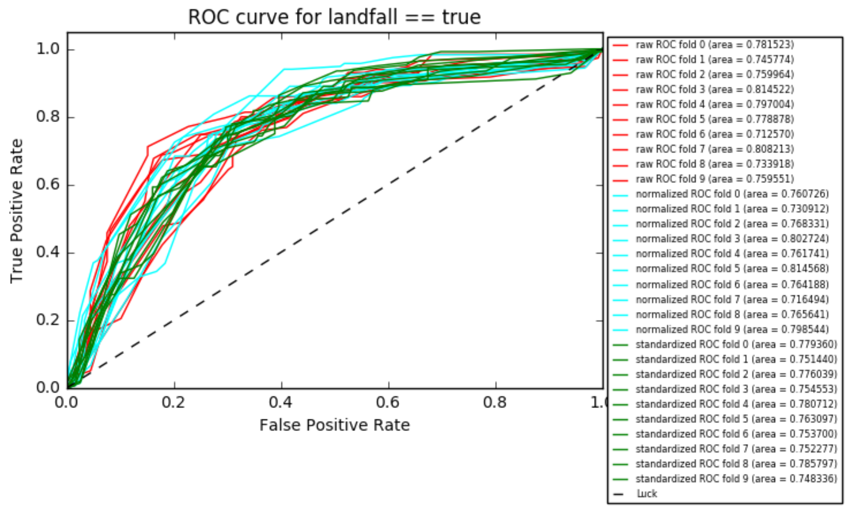
```
    # print metrics
    display(Markdown("**cross-validation scores ({} folds):**".format(folds)))
    table = "| **data** | **accuracy** | **mean AUC** |\n"
    table += "| --- | --- | --- |\n"
    table += "| raw | %0.2f (+/- %0.2f) | %f |\n" % (scores.mean(), scores.std() * 2, mean_auc_raw)
    table += "| normalized | %0.2f (+/- %0.2f) | %f |\n" % (scores_norm.mean(), scores_norm.std() * 2, mean_auc_norm)
    table += "| standardized | %0.2f (+/- %0.2f) | %f |\n" % (scores_std.mean(), scores_std.std() * 2, mean_auc_std)
    display(Markdown(table))


def print_timing(t0, t1, t2, train_size, test_size):
    """Print timing info of classifier training and prediction."""

    display(Markdown("**timing info:**"))
    table = "| **stage** | **sample size** | **execution time (s)** |\n"
    table += "| --- | --- | --- |\n"
    table += "| training on train set | %d | %f |\n" % (train_size, t1-t0)
    table += "| classifying test set | %d | %f |\n" % (test_size, t2-t1)
    table += "| total | -- | %f |\n" % (t2-t0)
    display(Markdown(table))


def train_and_classify(clf, X_train, y_train, X_test, y_test_truth):
    """This function encapsulates training of the classifier passed in
    using the training dataset, classification of the test dataset,
    and prints metrics on the classifier's performance."""

    # train classifier using the training dataset
    t0 = time.time()
    clf.fit(X_train, y_train)

    # predict on test dataset
    t1 = time.time()
    y_test_pred = clf.predict(X_test)
    t2 = time.time()

    # get class probabilities of positive class (landfall == True)
    y_scores = clf.predict_proba(X_test)[:,1]

    # print timing info
    print_timing(t0, t1, t2, len(X_train), len(X_test))

    # show ROC
    plot_roc(y_test_truth, y_scores)

    # print metrics
    print_metrics(y_test_truth, y_test_pred)

    # show confusion matrix
    cnf_matrix = metrics.confusion_matrix(y_test_truth, y_test_pred)
    plot_confusion_matrix(cnf_matrix, classes=CLASS_LABELS)


# extract training set features and class label
X_train = get_dummies(extract_features(trainset), CATEGORY_FEATURES)
y_train = extract_label(trainset)

# extract test set features and class label
X_test = get_dummies(extract_features(testset), CATEGORY_FEATURES)
y_test_truth = extract_label(testset).values

# normalized copy of train and test set (rescaled valued numeric attributes into the range of 0 and 1)
X_train_norm = preprocessing.normalize(X_train)
X_test_norm = preprocessing.normalize(X_test)

# standardized copy of train and test set (value distribution shifted to have a mean of zero and standard deviation of 1)
X_train_std = preprocessing.scale(X_train)
X_test_std = preprocessing.scale(X_test)
```

Decision Tree

DecisionTreeClassifier using Gini impurity

In [88]:

```
from sklearn import tree

# default decision tree classifier using Gini impurity
clf = tree.DecisionTreeClassifier(max_depth=5)

# cross validation
cross_validation_metrics(clf, trainset)
```



cross-validation scores (10 folds):

data	accuracy	mean AUC
raw	0.73 (+/- 0.04)	0.769184
normalized	0.70 (+/- 0.07)	0.768371
standardized	0.70 (+/- 0.08)	0.764527

In [89]:

```
# train and classify on raw datasets
train_and_classify(clf, X_train, y_train, X_test, y_test_truth)
```

timing info:

stage	sample size	execution time (s)
training on train set	1000	0.003143
classifying test set	3836	0.000689
total	--	0.003832

ROC:

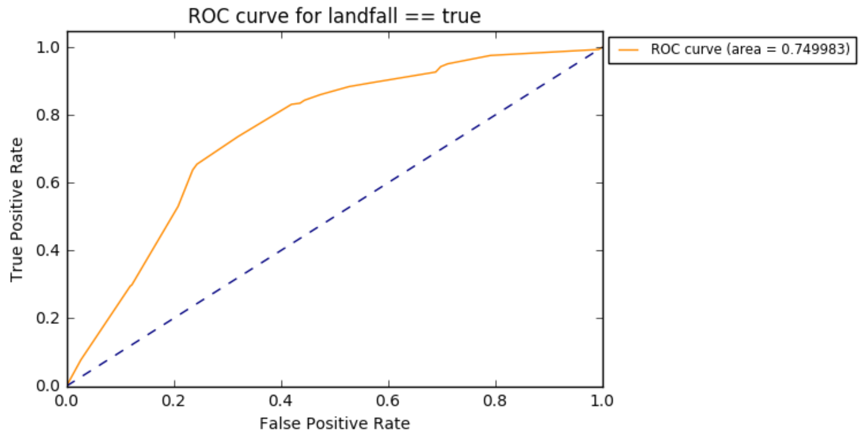
metric	score
AUC	0.749983

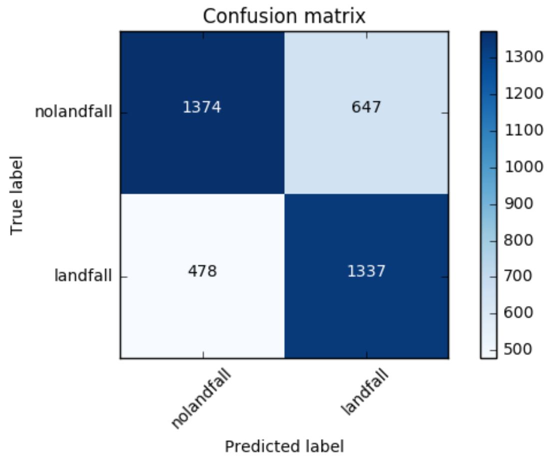
classification scores:

metric	score
accuracy_score	0.706726
average_precision_score	0.767570
f1_score	0.703869
recall_score	0.736639

classification report :

```
             precision    recall  f1-score   support

 nolandfall       0.74      0.68      0.71      2021
   landfall       0.67      0.74      0.70      1815

avg / total       0.71      0.71      0.71      3836
```





In [90]:

```
# train and classify on normalized datasets
train_and_classify(clf, X_train_norm, y_train, X_test_norm, y_test_truth)
```

timing info:

stage	sample size	execution time (s)
training on train set	1000	0.004656
classifying test set	3836	0.000604
total	--	0.005260

ROC:

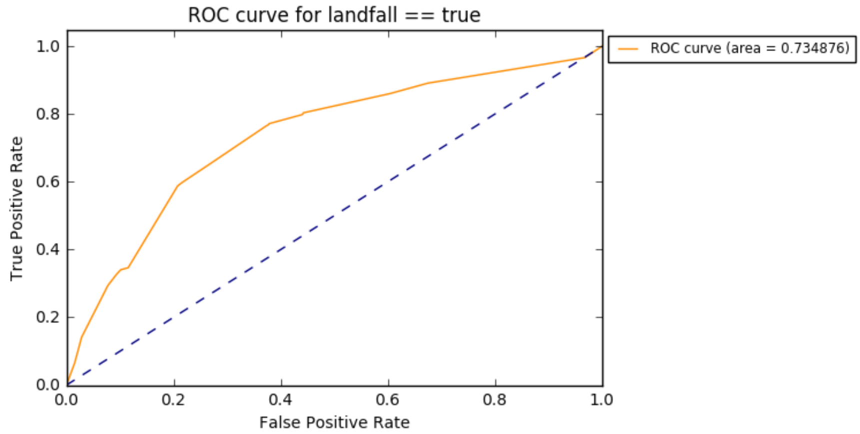
metric	score
AUC	0.734876

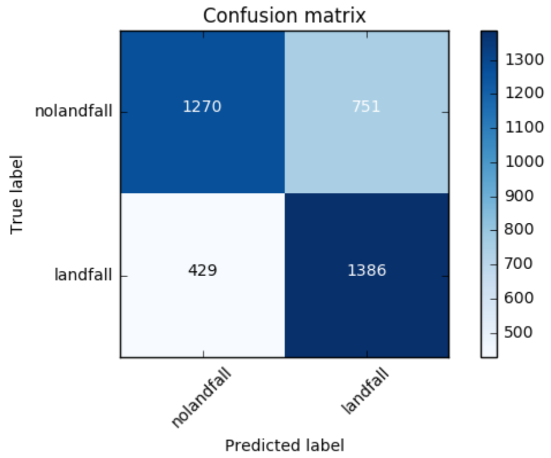
classification scores:

metric	score
accuracy_score	0.692388
average_precision_score	0.762022
f1_score	0.701417
recall_score	0.763636

classification report :

```
             precision    recall  f1-score   support

 nolandfall       0.75      0.63      0.68      2021
   landfall       0.65      0.76      0.70      1815

avg / total       0.70      0.69      0.69      3836
```





In [91]:

```
# train and classify on standardized datasets
train_and_classify(clf, X_train_std, y_train, X_test_std, y_test_truth)
```

timing info:

stage	sample size	execution time (s)
training on train set	1000	0.002879
classifying test set	3836	0.000438
total	--	0.003317

ROC:

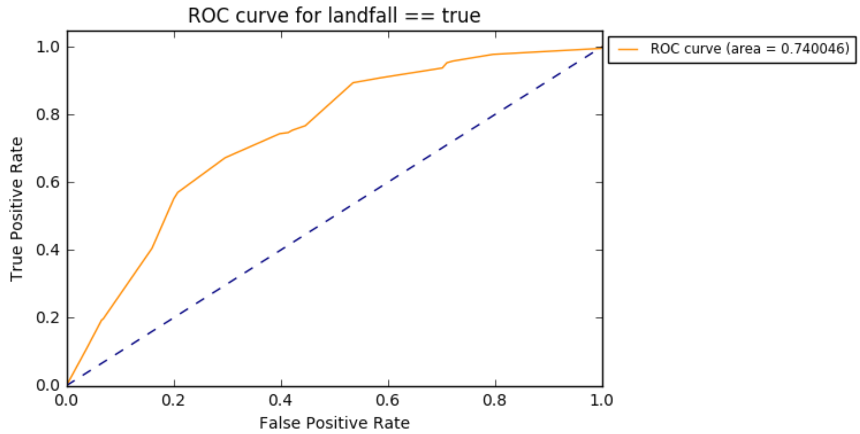
metric	score
AUC	0.740046

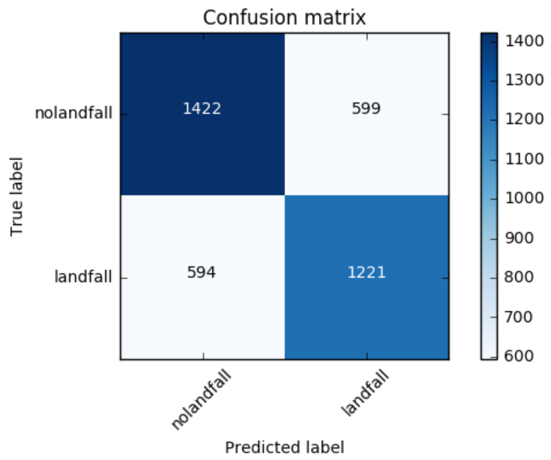
classification scores:

metric	score
accuracy_score	0.688999
average_precision_score	0.749228
f1_score	0.671802
recall_score	0.672727

classification report :

```
             precision    recall  f1-score   support

 nolandfall       0.71      0.70      0.70      2021
   landfall       0.67      0.67      0.67      1815

avg / total       0.69      0.69      0.69      3836
```

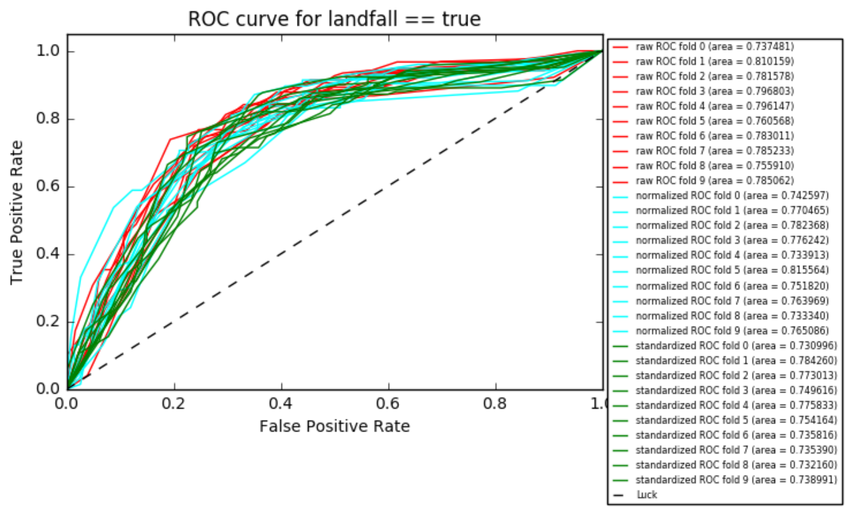




DecisionTreeClassifier using entropy for information gain

In [114]:

```
# decision tree classifier using entropy for information gain
clf = tree.DecisionTreeClassifier(criterion="entropy", max_depth=5)

# cross validation
cross_validation_metrics(clf, trainset)
```



cross-validation scores (10 folds):

data	accuracy	mean AUC
raw	0.73 (+/- 0.06)	0.779174
normalized	0.72 (+/- 0.05)	0.763495
standardized	0.70 (+/- 0.06)	0.750978

In [115]:

```
# train and classify on raw datasets
train_and_classify(clf, X_train, y_train, X_test, y_test_truth)
```

timing info:

stage	sample size	execution time (s)
training on train set	1000	0.003890
classifying test set	3836	0.000663
total	--	0.004553

ROC:

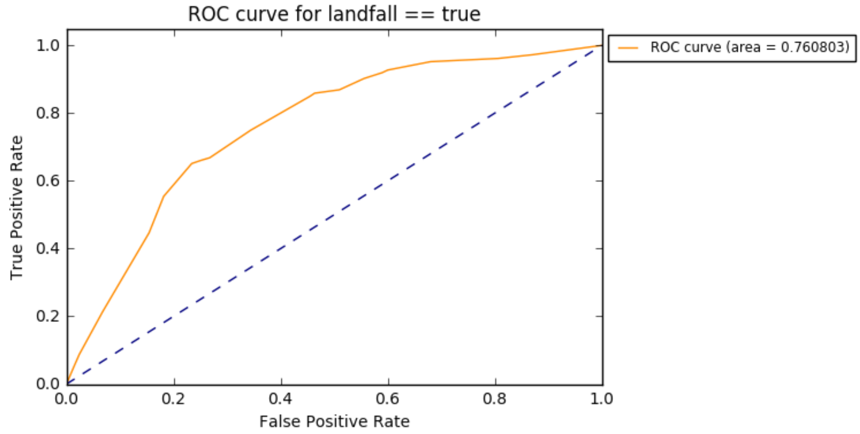
metric	score
AUC	0.760803

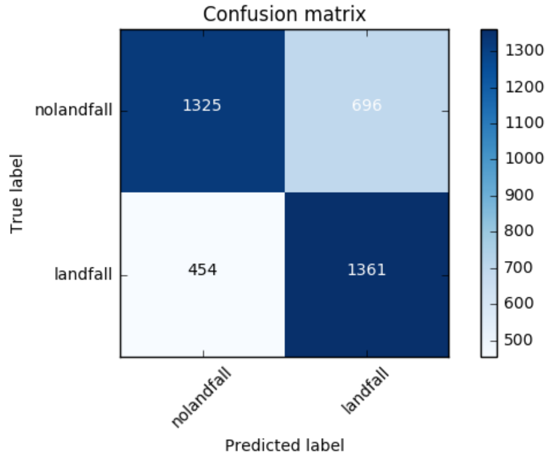
classification scores:

metric	score
accuracy_score	0.700209
average_precision_score	0.764929
f1_score	0.702996
recall_score	0.749862

classification report :

```
             precision    recall  f1-score   support

 nolandfall       0.74      0.66      0.70      2021
   landfall       0.66      0.75      0.70      1815

avg / total       0.71      0.70      0.70      3836
```





In [116]:

```
# train and classify on normalized datasets
train_and_classify(clf, X_train_norm, y_train, X_test_norm, y_test_truth)
```

timing info:

stage	sample size	execution time (s)
training on train set	1000	0.005998
classifying test set	3836	0.000409
total	--	0.006407

ROC:

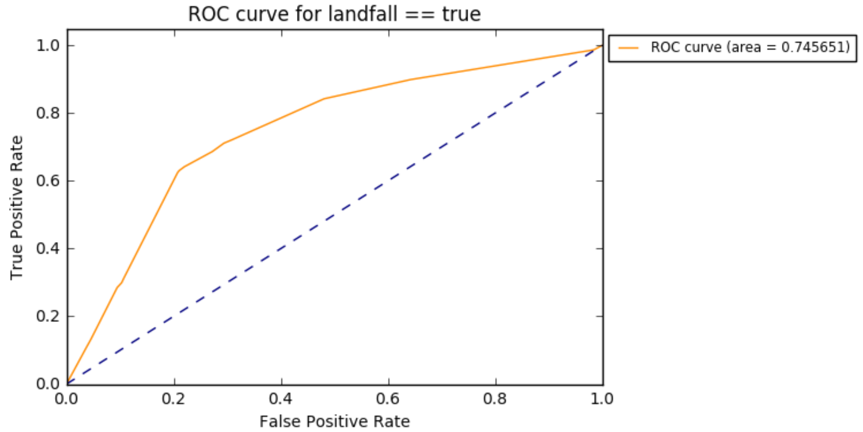
metric	score
AUC	0.745651

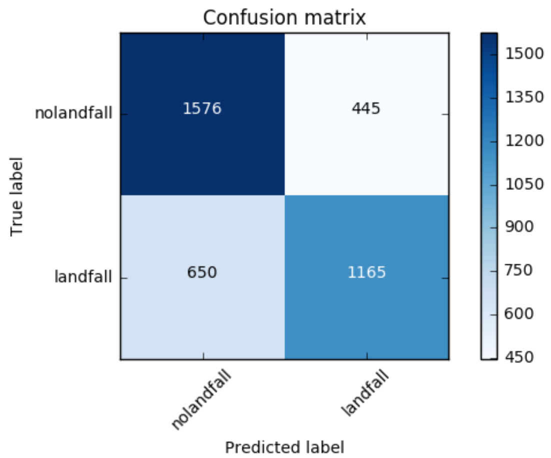
classification scores:

metric	score
accuracy_score	0.714546
average_precision_score	0.767462
f1_score	0.680292
recall_score	0.641873

classification report :

```
             precision    recall  f1-score   support

 nolandfall       0.71      0.78      0.74      2021
   landfall       0.72      0.64      0.68      1815

avg / total       0.72      0.71      0.71      3836
```





In [117]:

```
# train and classify on standardized datasets
train_and_classify(clf, X_train_std, y_train, X_test_std, y_test_truth)
```

timing info:

stage	sample size	execution time (s)
training on train set	1000	0.004248
classifying test set	3836	0.000765
total	--	0.005013

ROC:

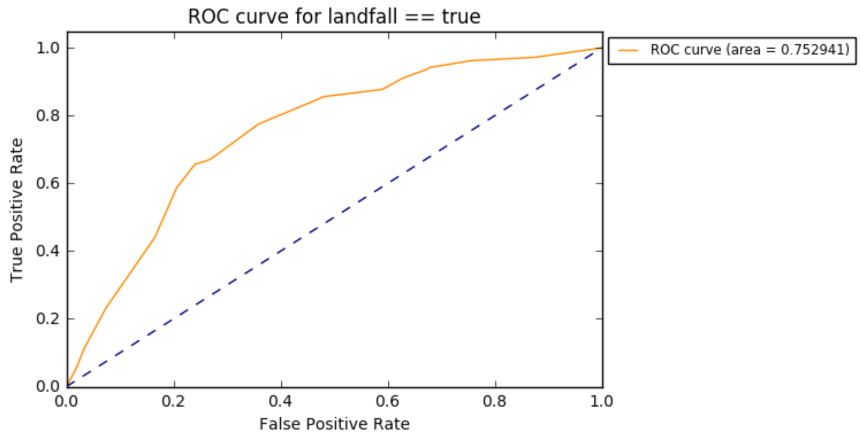
metric	score
AUC	0.752941

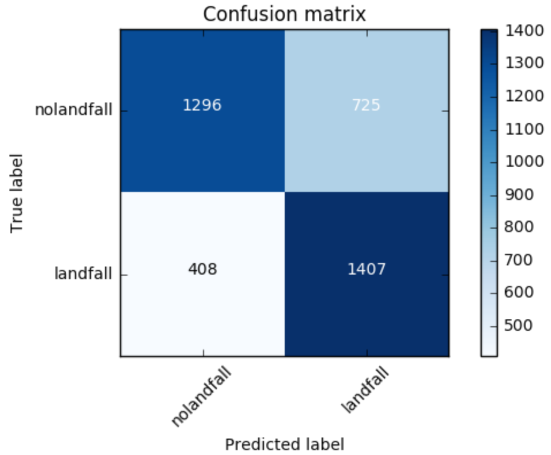
classification scores:

metric	score
accuracy_score	0.704640
average_precision_score	0.770756
f1_score	0.712947
recall_score	0.775207

classification report :

```
             precision    recall  f1-score   support

 nolandfall       0.76      0.64      0.70      2021
   landfall       0.66      0.78      0.71      1815

avg / total       0.71      0.70      0.70      3836
```





Decision Tree Analysis

In exploring the Decision Tree learning algorithm in scikit-learn, we ran the training of the classifier using 2 different classification criterion: Gini impurity and cross-entropy. Also, by default the DecisionTreeClassifier in scikit-learn doesn't limit the depth of the tree and thus will expand nodes until all leaves are pure or all leaves contain less than a minimum number of samples (2 by default). Since we want to avoid overfitting, we want to limit the maximum depth of the tree to the number of features (8) in the training set. Thus using the cross_validation_metrics() function we defined, we can iterate over tweaking the **max_depth** parameter to see at what depth, between 1-8, the model performs optimally.

As you can see above, I found that for both DecisionTreeClassifier instances, one using Gini impurity and the other using entropy for information gain, the parameter **max_depth=5** yields the best performance in terms of accuracy and AUC (area under the ROC curve) for all cases of the datasets (raw, normalized, and standardized).

The 10-fold ShuffleSort cross-validation run of the classifier using Gini impurity showed that running it on the unscaled raw training set yields the best performance in terms of both accuracy and AUC:

data	accuracy	mean AUC
raw	0.73 (+/- 0.04)	0.769184
normalized	0.70 (+/- 0.07)	0.768371
standardized	0.70 (+/- 0.08)	0.764527

and similarly for the 10-fold ShuffleSort cross-validation run of the classifier using entropy for information gain:

data	accuracy	mean AUC
raw	0.73 (+/- 0.06)	0.779174
normalized	0.72 (+/- 0.05)	0.763495
standardized	0.70 (+/- 0.06)	0.750978

When we run the classifiers on the actual test set, we get accuracy and AUC scores that validate the scores we received in our cross-validation exercise. The following table aggregates the accuracy and AUC scores from the above classification runs on the test set:

criterion	data	accuracy	AUC
gini	raw	0.706726	0.749983
gini	normalized	0.692388	0.734876
gini	standardized	0.688999	0.740046
entropy	raw	0.700209	0.760803
entropy	normalized	0.714546	0.745651
entropy	standardized	0.704640	0.752941

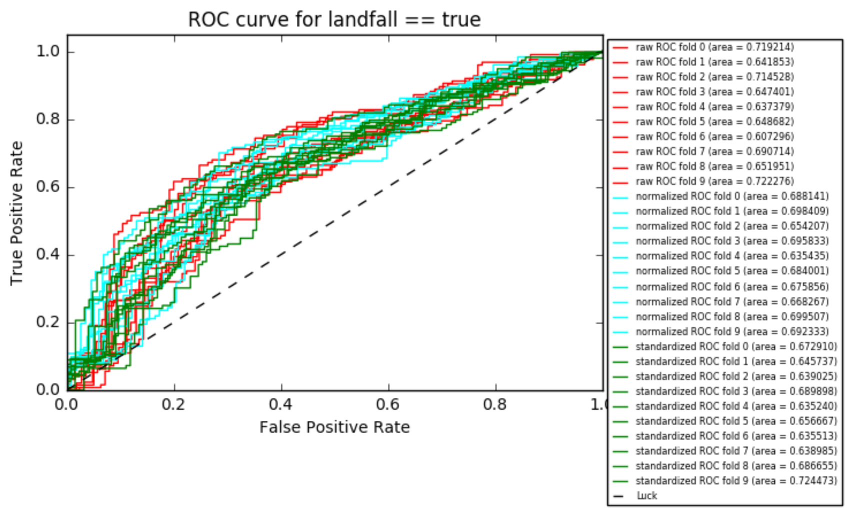
We can see from the accuracy and AUC scores that the classifier using entropy for information gain generally performed better on the test set than the classifier using Gini impurity for classification. In the cross-validation scores, using the raw training set yielded the best accuracy and AUC score. However the prediction scores on the test set show that accuracy was best when run on the normalized test set while AUC was best when run on the raw test set. In our case, the cost associated with minimizing false negatives is far greater than accuracy and so we should prefer the entropy-based DecisionTreeClassifier to be run on our raw data set.

Naive Bayes

Gaussian Naive Bayes classifier where the likelihood of the features is assumed to be Gaussian

In [18]:

```
from sklearn import naive_bayes

# Gaussian Naive Bayes classifier where the likelihood of the features is assumed to be Gaussian
clf = naive_bayes.GaussianNB()

# cross validation
cross_validation_metrics(clf, trainset)
```



cross-validation scores (10 folds):

data	accuracy	mean AUC
raw	0.55 (+/- 0.06)	0.668165
normalized	0.60 (+/- 0.05)	0.679350
standardized	0.60 (+/- 0.04)	0.662281

In [19]:

```
# train and classify on raw datasets
train_and_classify(clf, X_train, y_train, X_test, y_test_truth)
```

timing info:

stage	sample size	execution time (s)
training on train set	1000	0.002317
classifying test set	3836	0.002176
total	--	0.004493

ROC:

metric	score
AUC	0.666868

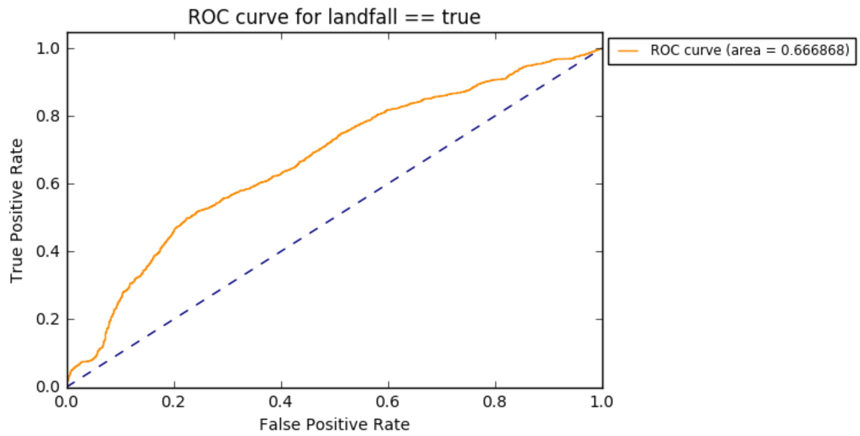
classification scores:

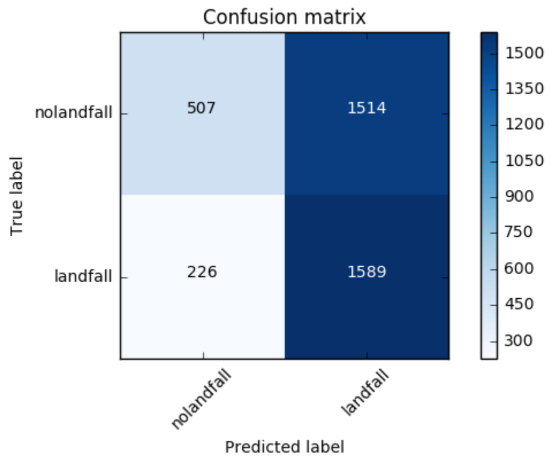
metric	score
accuracy_score	0.546403
average_precision_score	0.723241
f1_score	0.646198
recall_score	0.875482

classification report :

```
             precision    recall  f1-score   support

 nolandfall       0.69      0.25      0.37      2021
```

```
  landfall       0.51      0.88      0.65      1815

avg / total       0.61      0.55      0.50      3836
```





In [20]:

```
# train and classify on normalized datasets
train_and_classify(clf, X_train_norm, y_train, X_test_norm, y_test_truth)
```

timing info:

stage	sample size	execution time (s)
training on train set	1000	0.001949
classifying test set	3836	0.001357
total	--	0.003306

ROC:

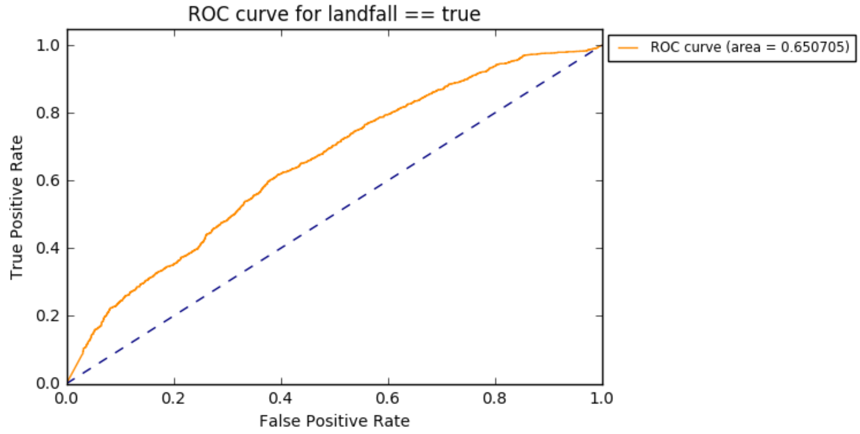
metric	score
AUC	0.650705

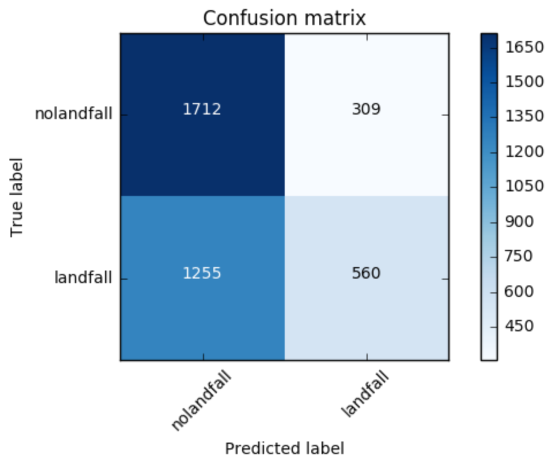
classification scores:

metric	score
accuracy_score	0.592284
average_precision_score	0.640061
f1_score	0.417288
recall_score	0.308540

classification report:

```
             precision    recall  f1-score   support

 nolandfall       0.58      0.85      0.69      2021
   landfall       0.64      0.31      0.42      1815

avg / total       0.61      0.59      0.56      3836
```





In [21]:

```
# train and classify on standardized datasets
train_and_classify(clf, X_train_std, y_train, X_test_std, y_test_truth)
```

timing info:

stage	sample size	execution time (s)
training on train set	1000	0.001816
classifying test set	3836	0.001492
total	--	0.003308

ROC:

metric	score
AUC	0.500000

```
/Users/gmanipon/anaconda3/lib/python3.5/site-packages/sklearn/metrics/classification.py:1113: UndefinedMetricWarning: F-score is ill-defined and bei
ng set to 0.0 due to no predicted samples.
  'precision', 'predicted', average, warn_for)
```

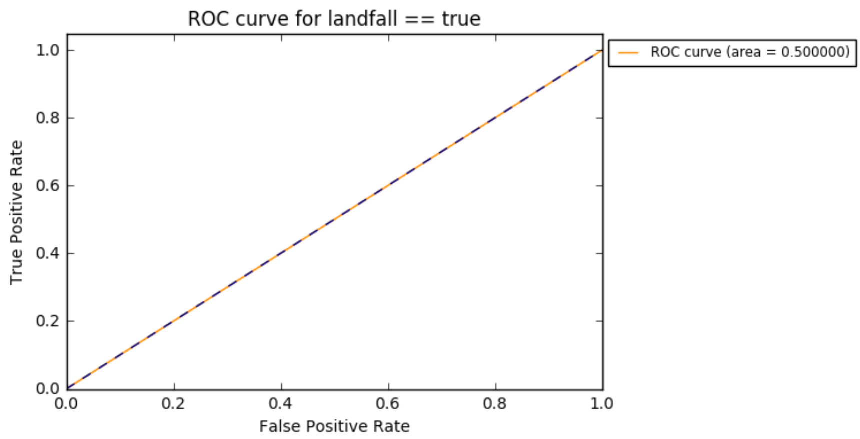
classification scores:

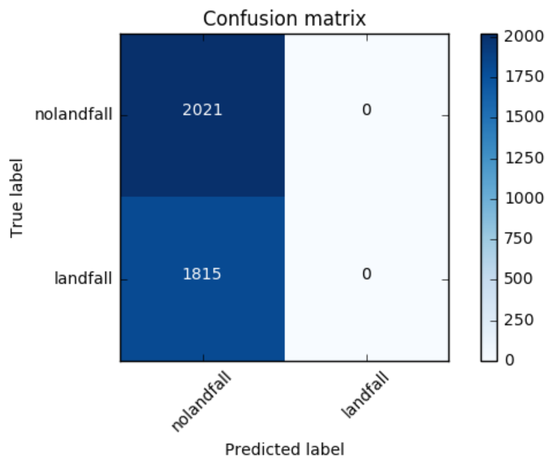
metric	score
accuracy_score	0.526851
average_precision_score	0.736575
f1_score	0.000000
recall_score	0.000000

classification report:

```
/Users/gmanipon/anaconda3/lib/python3.5/site-packages/sklearn/metrics/classification.py:1113: UndefinedMetricWarning: Precision and F-score are ill-
defined and being set to 0.0 in labels with no predicted samples.
  'precision', 'predicted', average, warn_for)
```

```
             precision    recall  f1-score   support

 nolandfall       0.53      1.00      0.69      2021
   landfall       0.00      0.00      0.00      1815

avg / total       0.28      0.53      0.36      3836
```

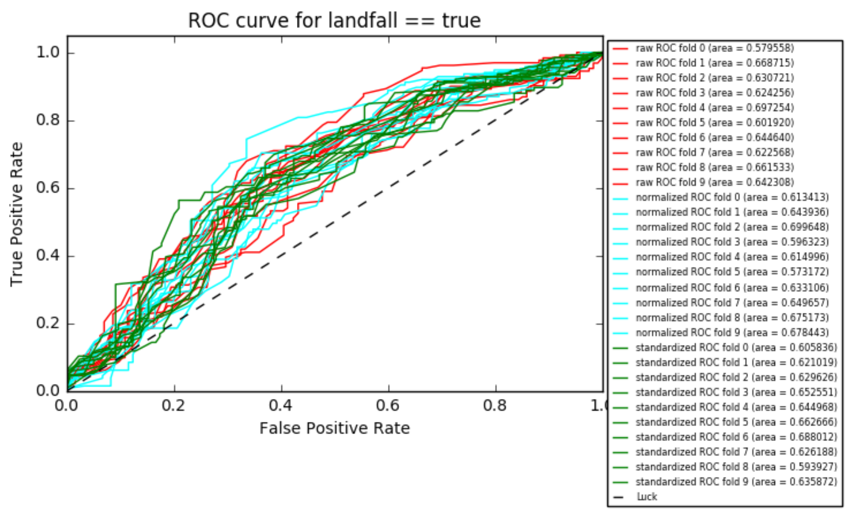




Bernoulli Naive Bayes classifier for data that is distributed according to multivariate Bernoulli distributions

In [22]:

```
# Bernoulli Naive Bayes classifier for data that is distributed according to multivariate Bernoulli distributions
clf = naive_bayes.BernoulliNB()

# cross validation
cross_validation_metrics(clf, trainset)
```



cross-validation scores (10 folds):

data	accuracy	mean AUC
raw	0.62 (+/- 0.07)	0.637157
normalized	0.61 (+/- 0.05)	0.637834
standardized	0.62 (+/- 0.08)	0.635853

In [23]:

```
# train and classify on raw datasets
train_and_classify(clf, X_train, y_train, X_test, y_test_truth)
```

timing info:

stage	sample size	execution time (s)
training on train set	1000	0.012317
classifying test set	3836	0.002052
total	--	0.014369

ROC:

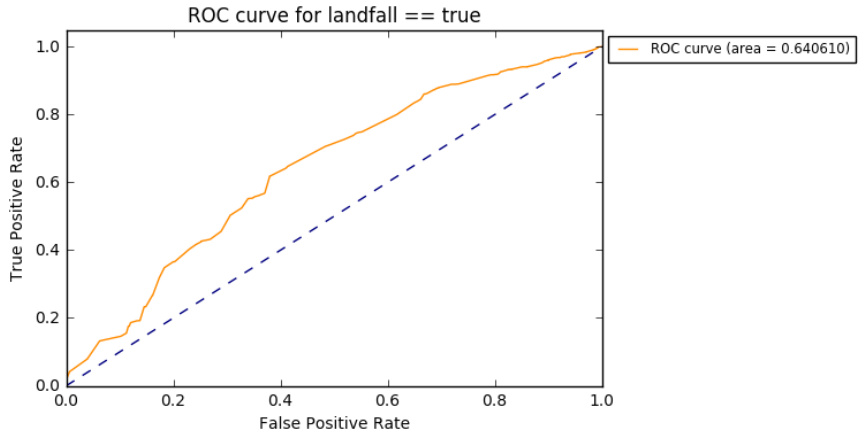
metric	score
AUC	0.640610

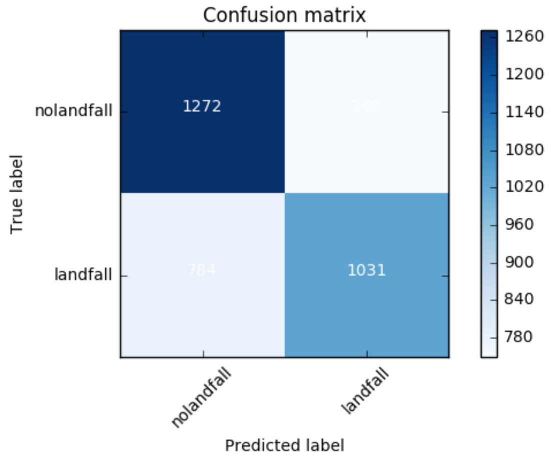
classification scores:

metric	score
accuracy_score	0.600365
average_precision_score	0.675819
f1_score	0.573574
recall_score	0.568044

classification report :

```
             precision    recall  f1-score   support

 nolandfall       0.62      0.63      0.62      2021
   landfall       0.58      0.57      0.57      1815

avg / total       0.60      0.60      0.60      3836
```





In [24]:

```
# train and classify on normalized datasets
train_and_classify(clf, X_train_norm, y_train, X_test_norm, y_test_truth)
```

timing info:

stage	sample size	execution time (s)
training on train set	1000	0.002476
classifying test set	3836	0.001796
total	--	0.004272

ROC:

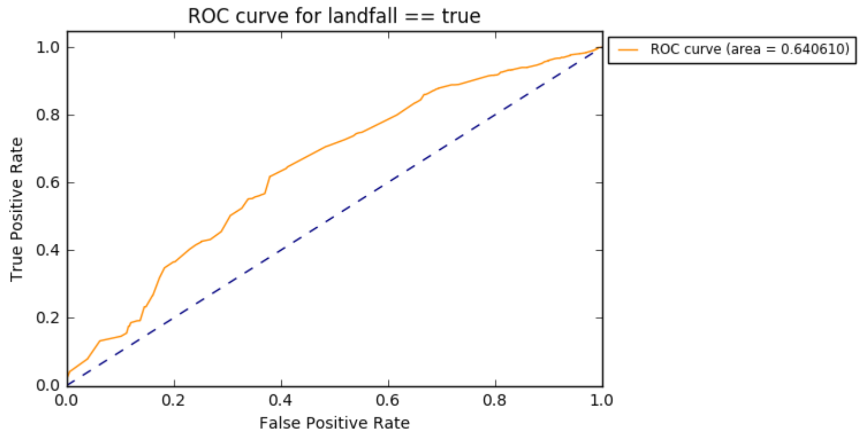
metric	score
AUC	0.640610

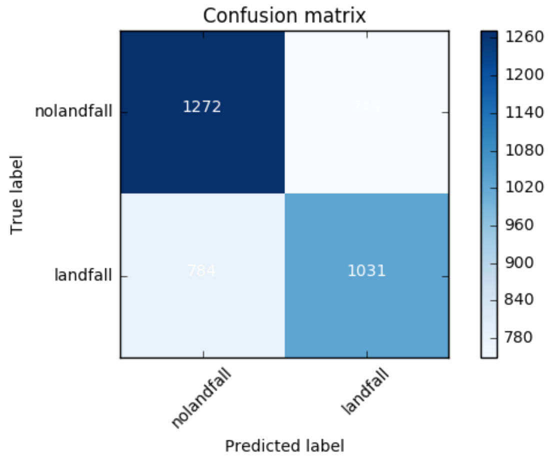
classification scores:

metric	score
accuracy_score	0.600365
average_precision_score	0.675819
f1_score	0.573574
recall_score	0.568044

classification report :

```
             precision    recall  f1-score   support

 nolandfall       0.62      0.63      0.62      2021
   landfall       0.58      0.57      0.57      1815

avg / total       0.60      0.60      0.60      3836
```





In [25]:

```
# train and classify on standardized datasets
train_and_classify(clf, X_train_std, y_train, X_test_std, y_test_truth)
```

timing info:

stage	sample size	execution time (s)
training on train set	1000	0.002256
classifying test set	3836	0.001820
total	--	0.004076

ROC:

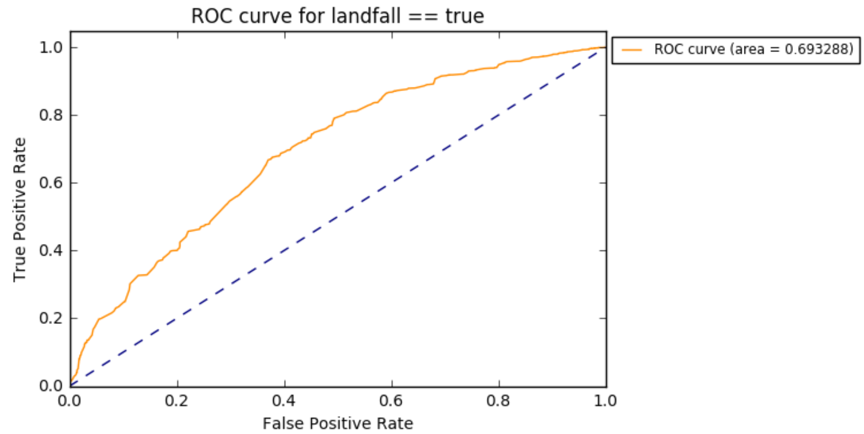
metric	score
AUC	0.693288

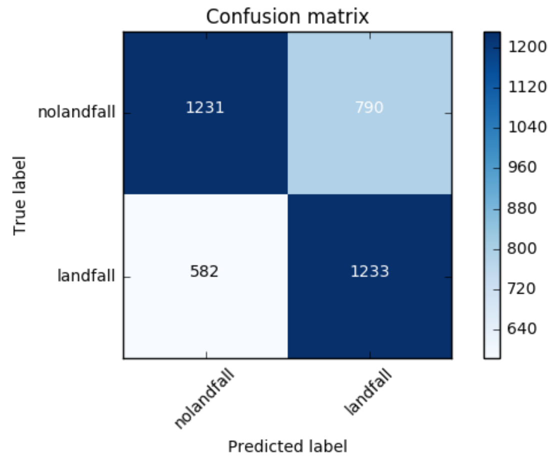
classification scores:

metric	score
accuracy_score	0.642336
average_precision_score	0.720275
f1_score	0.642522
recall_score	0.679339

classification report :

```
             precision    recall  f1-score   support

 nolandfall       0.68      0.61      0.64      2021
   landfall       0.61      0.68      0.64      1815

avg / total       0.65      0.64      0.64      3836
```





Naive Bayes Analysis

In exploring the Naive Bayes learning algorithm in scikit-learn, we ran the training of the classifier using 2 different classifiers: Gaussian and Bernoulli. The GaussianNB classifier implements the Gaussian Naive Bayes algorithm for classification where the likelihood of the features is assumed to be Gaussian. The BernoulliNB classifier implements the Naive Bayes training and classification algorithms for data that is distributed according to multivariate Bernoulli distributions.

The 10-fold ShuffleSort cross-validation run of the Gaussian Naive Bayes classifier showed that running it on the normalized training set yields the best performance in terms of both accuracy and AUC:

data	accuracy	mean AUC
raw	0.55 (+/- 0.06)	0.668165
normalized	0.60 (+/- 0.05)	0.679350
standardized	0.60 (+/- 0.04)	0.662281

and for the 10-fold ShuffleSort cross-validation run of the Bernoulli Naive Bayes classifier the run on the normalized training set yields the best AUC score:

data	accuracy	mean AUC
raw	0.62 (+/- 0.07)	0.637157
normalized	0.61 (+/- 0.05)	0.637834
standardized	0.62 (+/- 0.08)	0.635853

When we run the classifiers on the actual test set, we get accuracy and AUC scores that validate the scores we received in our cross-validation exercise (with the exception of the Gaussian classifier run on the standardized test set). The following table aggregates the accuracy and AUC scores from the above classification runs on the test set:

NB type	data	accuracy	AUC
gaussian	raw	0.546403	0.666868
gaussian	normalized	0.592284	0.650705
gaussian	standardized	0.526851	0.500000
bernoulli	raw	0.600365	0.640610
bernoulli	normalized	0.600365	0.640610
bernoulli	standardized	0.642336	0.693288

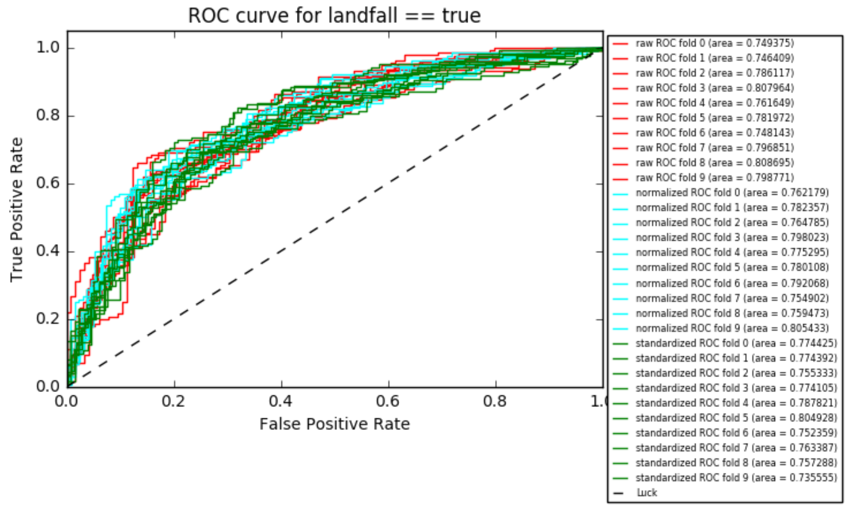
Overall, the Bernoulli classifier performed better than the Gaussian classifier on the test set in regards to accuracy as indicated by the cross-validation scores. In terms of AUC scores, the results show a discrepancy. The cross-validation scores indicate that the Gaussian classifier run on the normalized training set should yield the best AUC score. However the results above show that the best performer on the test set in terms of both accuracy and AUC is the Bernoulli classifier run on the standardized test set. In our case, we hold that the cost associated with minimizing false negatives is far greater than accuracy however by choosing the Bernoulli classifier we get the best of both worlds when the training and test data sets are standardized.

Logistic Regression

In [26]:

```
from sklearn import linear_model

# train the classifier
clf = linear_model.LogisticRegression()
```

```
# cross validation
cross_validation_metrics(clf, trainset)
```



cross-validation scores (10 folds):

data	accuracy	mean AUC
raw	0.70 (+/- 0.04)	0.778408
normalized	0.65 (+/- 0.09)	0.777821
standardized	0.66 (+/- 0.05)	0.767900

In [27]:

```
# train and classify on raw datasets
train_and_classify(clf, X_train, y_train, X_test, y_test_truth)
```

timing info:

stage	sample size	execution time (s)
training on train set	1000	0.005730
classifying test set	3836	0.001707
total	--	0.007437

ROC:

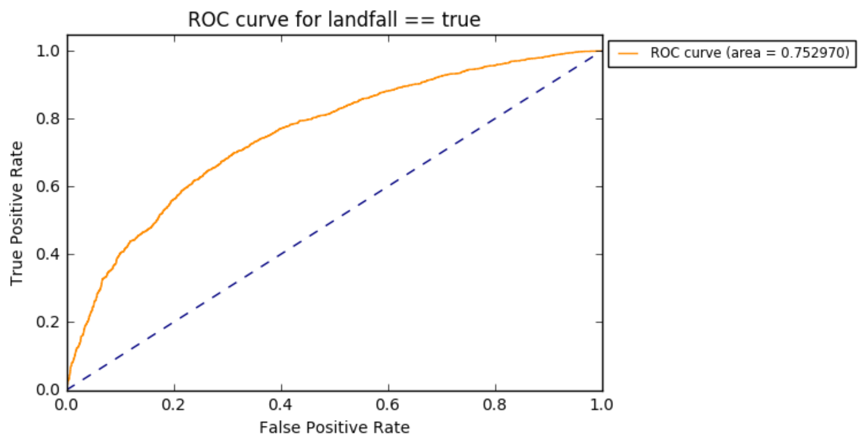
metric	score
AUC	0.752970

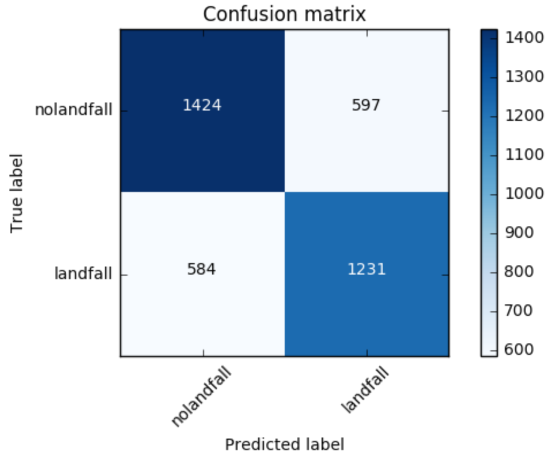
classification scores:

metric	score
accuracy_score	0.692127
average_precision_score	0.751946
f1_score	0.675817
recall_score	0.678237

classification report :

```
             precision    recall  f1-score   support

 nolandfall       0.71      0.70      0.71      2021
   landfall       0.67      0.68      0.68      1815

avg / total       0.69      0.69      0.69      3836
```





In [28]:

```
# train and classify on normalized datasets
train_and_classify(clf, X_train_norm, y_train, X_test_norm, y_test_truth)
```

timing info:

stage	sample size	execution time (s)
training on train set	1000	0.002733
classifying test set	3836	0.000263
total	--	0.002996

ROC:

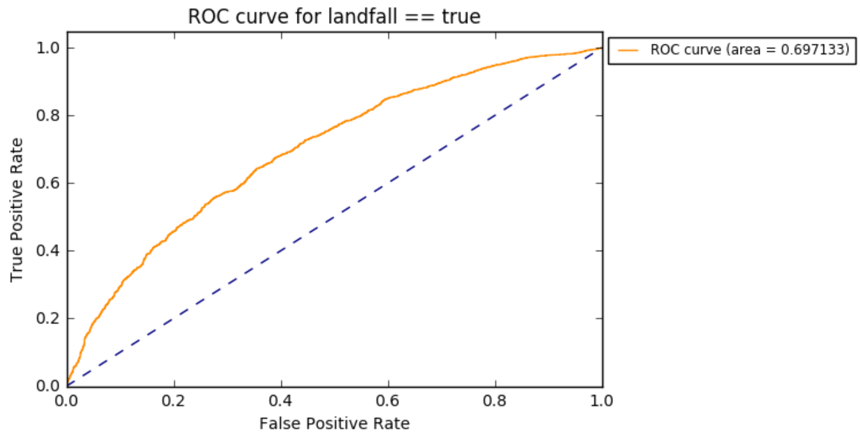
metric	score
AUC	0.697133

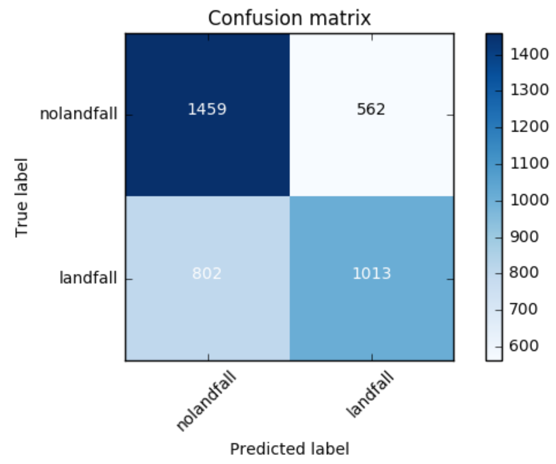
classification scores:

metric	score
accuracy_score	0.644421
average_precision_score	0.705187
f1_score	0.597640
recall_score	0.558127

classification report :

```
             precision    recall  f1-score   support

 nolandfall       0.65      0.72      0.68      2021
   landfall       0.64      0.56      0.60      1815

avg / total       0.64      0.64      0.64      3836
```





In [29]:

```
# train and classify on standardized datasets
train_and_classify(clf, X_train_std, y_train, X_test_std, y_test_truth)
```

timing info:

stage	sample size	execution time (s)
training on train set	1000	0.004617
classifying test set	3836	0.000286
total	--	0.004903

ROC:

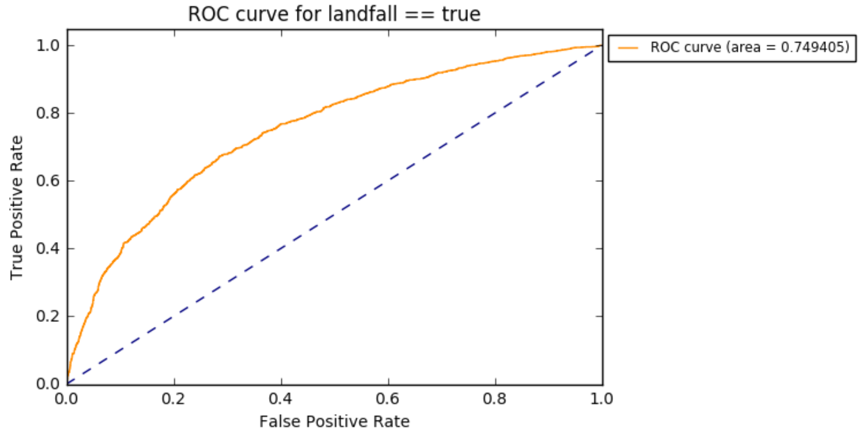
metric	score
AUC	0.749405

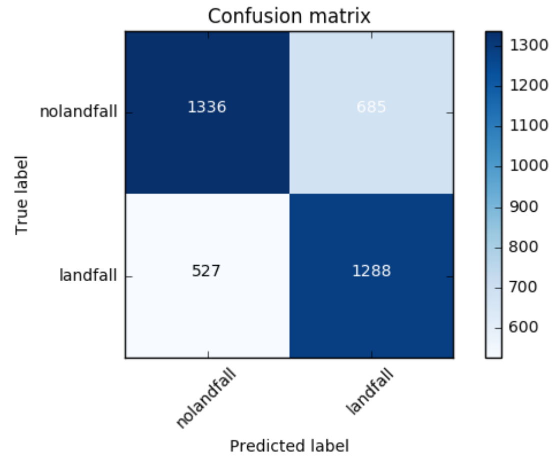
classification scores:

metric	score
accuracy_score	0.684046
average_precision_score	0.749919
f1_score	0.680042
recall_score	0.709642

classification report :

```
             precision    recall  f1-score   support

 nolandfall       0.72      0.66      0.69      2021
   landfall       0.65      0.71      0.68      1815

avg / total       0.69      0.68      0.68      3836
```





Logistic Regression Analysis

The 10-fold ShuffleSort cross-validation run of the Logistic Regression classifier showed that running it on the raw training set yields the best performance in terms of both accuracy and AUC:

data	accuracy	mean AUC
raw	0.70 (+/- 0.04)	0.778408
normalized	0.65 (+/- 0.09)	0.777821
standardized	0.66 (+/- 0.05)	0.767900

The following table aggregates the accuracy and AUC scores from the above classification runs on the test set:

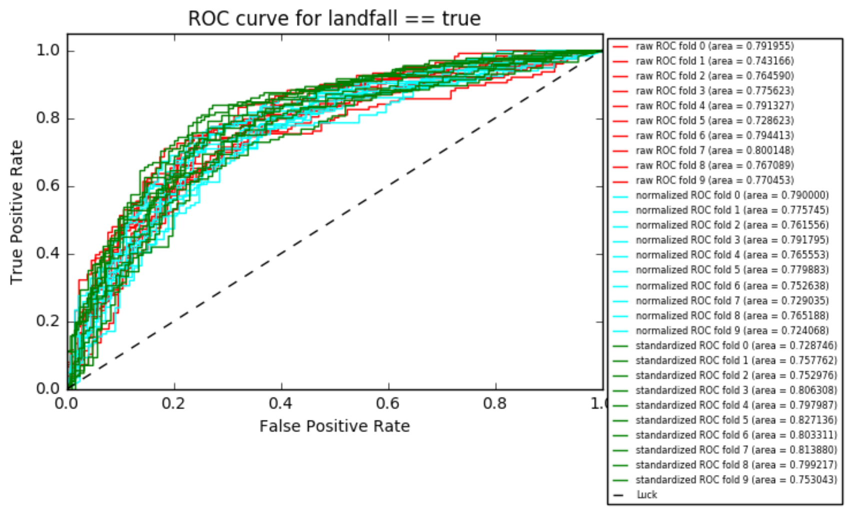
data	accuracy	AUC
raw	0.692127	0.752970
normalized	0.644421	0.697133
standardized	0.684046	0.749405

Here the choice is clear. The best performance in terms of accuracy and AUC comes from running the Logistic Regression classifier on the raw training and test data sets.

Nearest Neighbor

In [341]:

```
from sklearn import neighbors

# train the classifier
n_neighbors = 15
clf = neighbors.KNeighborsClassifier(n_neighbors, weights="distance")

# cross validation
cross_validation_metrics(clf, trainset)
```



cross-validation scores (10 folds):

data	accuracy	mean AUC
raw	0.72 (+/- 0.02)	0.772710
normalized	0.68 (+/- 0.03)	0.763894
standardized	0.71 (+/- 0.05)	0.784250

In [342]:

```
# train and classify on raw datasets
```

```
train_and_classify(clf, X_train, y_train, X_test, y_test_truth)
```

timing info:

stage	sample size	execution time (s)
training on train set	1000	0.002118
classifying test set	3836	0.041866
total	--	0.043984

ROC:

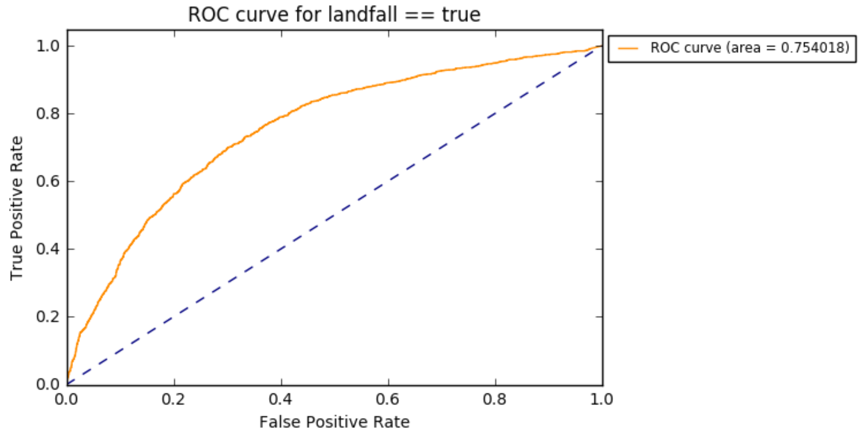
metric	score
AUC	0.754018

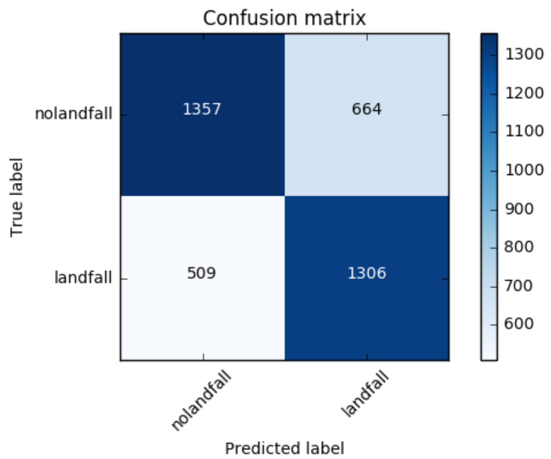
classification scores:

metric	score
accuracy_score	0.694213
average_precision_score	0.757597
f1_score	0.690092
recall_score	0.719559

classification report :

```
             precision    recall  f1-score   support

 nolandfall       0.73      0.67      0.70      2021
   landfall       0.66      0.72      0.69      1815

avg / total       0.70      0.69      0.69      3836
```





In [343]:

```
# train and classify on normalized datasets
train_and_classify(clf, X_train_norm, y_train, X_test_norm, y_test_truth)
```

timing info:

stage	sample size	execution time (s)
training on train set	1000	0.001930
classifying test set	3836	0.032172
total	--	0.034102

ROC:

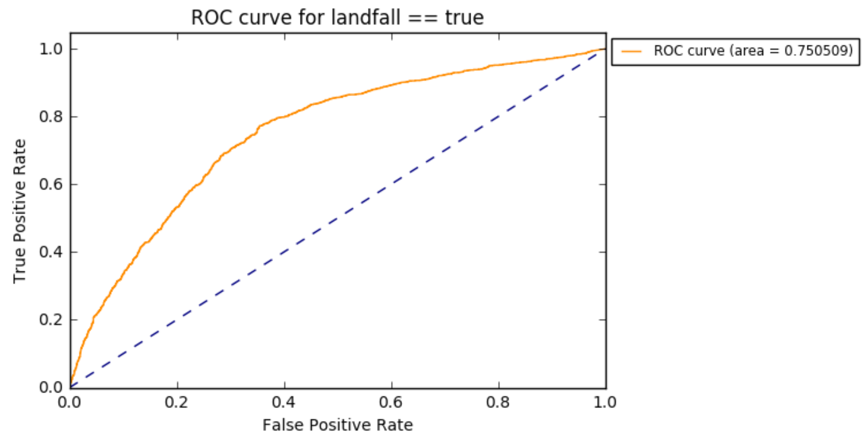
metric	score
AUC	0.750509

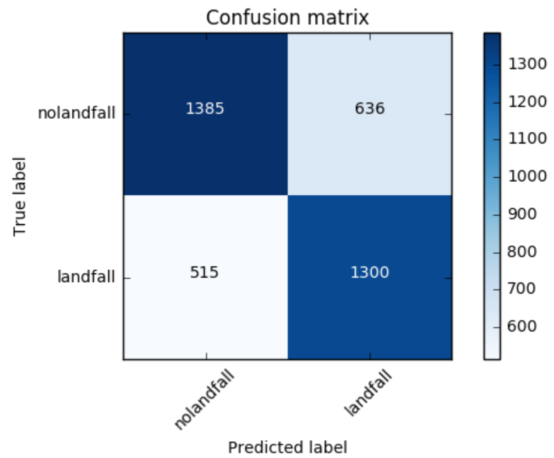
classification scores:

metric	score
accuracy_score	0.699948
average_precision_score	0.760998
f1_score	0.693148
recall_score	0.716253

classification report :

```
             precision    recall  f1-score   support

 nolandfall       0.73      0.69      0.71      2021
   landfall       0.67      0.72      0.69      1815

avg / total       0.70      0.70      0.70      3836
```





In [344]:

```
# train and classify on standardized datasets
train_and_classify(clf, X_train_std, y_train, X_test_std, y_test_truth)
```

timing info:

stage	sample size	execution time (s)
training on train set	1000	0.003015
classifying test set	3836	0.248369
total	--	0.251384

ROC:

metric	score
AUC	0.763377

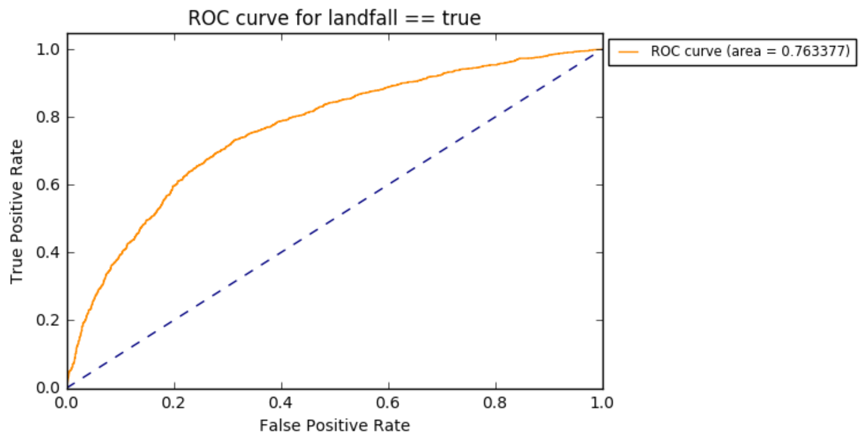
classification scores:

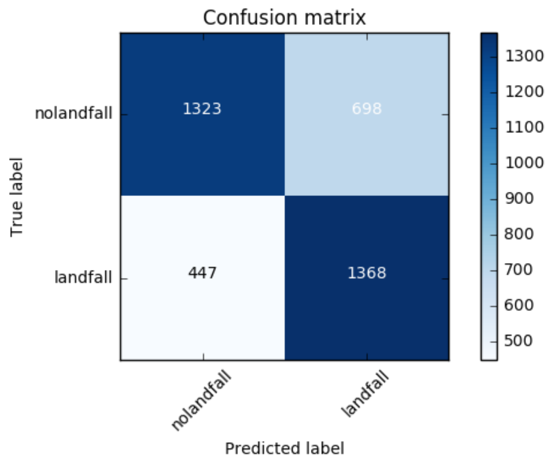
metric	score
accuracy_score	0.701512
average_precision_score	0.766198
f1_score	0.704973
recall_score	0.753719

classification report :

```
             precision    recall  f1-score   support

 nolandfall       0.75      0.65      0.70      2021
   landfall       0.66      0.75      0.70      1815
```

```
avg / total       0.71      0.70      0.70      3836
```





Nearest Neighbor Analysis

In exploring the Nearest Neighbor learning algorithm in scikit-learn, we ran the training of the classifier using 2 different weight functions: "uniform" where all points in each neighborhood are weighted equally and "distance" where close neighbors have greater influence than neighbors further away. Also, by default the KNeighborsClassifier in scikit-learn sets the default number of neighbors to use to 5. Thus using the cross_validation_metrics() function we defined, we can iterate over tweaking the n_neighbors and weights parameter to find the settings that perform best on our training set.

As you can see above, I found that the parameters n_neighbors=15 and weights=distance yield the best performance in terms of accuracy and AUC (area under the ROC curve) for all cases of the datasets (raw, normalized, and standardized).

The 10-fold ShuffleSort cross-validation run of the Nearest Neighbor classifier showed that running it on the standardized training set yields the best performance in terms of AUC:

data	accuracy	mean AUC
raw	0.72 (+/- 0.02)	0.772710
normalized	0.68 (+/- 0.03)	0.763894
standardized	0.71 (+/- 0.05)	0.784250

The following table aggregates the accuracy and AUC scores from the above classification runs on the test set:

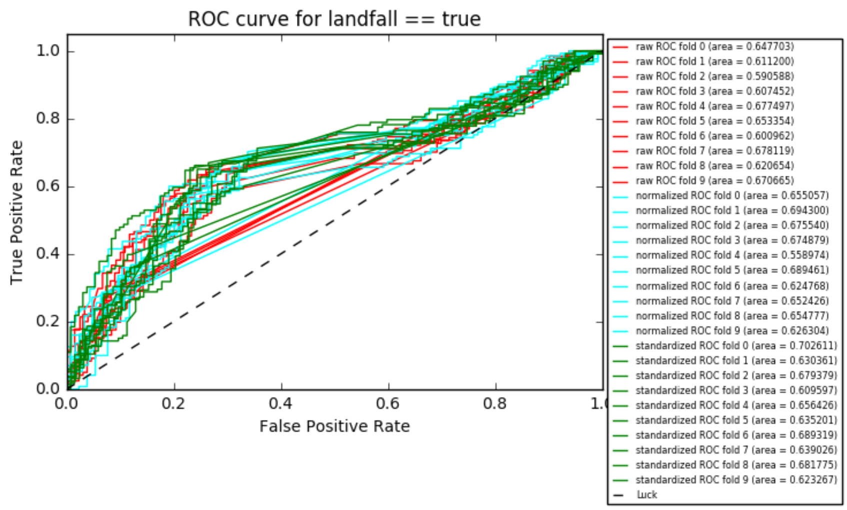
data	accuracy	AUC
raw	0.694213	0.754018
normalized	0.699948	0.750509
standardized	0.701512	0.763377

Here the choice is clear. The best performance in terms of accuracy and AUC comes from running the Nearest Neighbor classifier on the standardized training and test data sets.

Support Vector Machines

In [347]:

```
from sklearn import svm

# train the classifier
clf = svm.SVC(probability=True)

# cross validation
cross_validation_metrics(clf, trainset)
```



cross-validation scores (10 folds):

data	accuracy	mean AUC
raw	0.53 (+/- 0.09)	0.635947
normalized	0.64 (+/- 0.05)	0.650702
standardized	0.62 (+/- 0.04)	0.654708

In [348]:

```
# train and classify on raw datasets
train_and_classify(clf, X_train, y_train, X_test, y_test_truth)
```

timing info:

stage	sample size	execution time (s)
training on train set	1000	0.219597
classifying test set	3836	0.113664
total	--	0.333261

ROC:

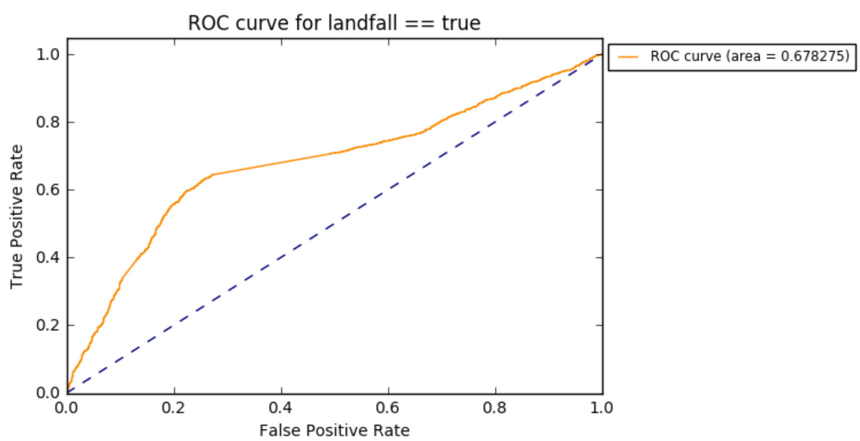
metric	score
AUC	0.678275

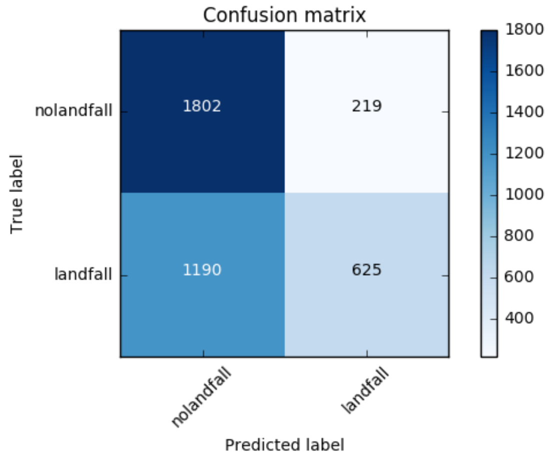
classification scores:

metric	score
accuracy_score	0.632690
average_precision_score	0.697546
f1_score	0.470102
recall_score	0.344353

classification report:

```
             precision    recall  f1-score   support

 nolandfall       0.60      0.89      0.72      2021
   landfall       0.74      0.34      0.47      1815

avg / total       0.67      0.63      0.60      3836
```





In [349]:

```
# train and classify on normalized datasets
train_and_classify(clf, X_train_norm, y_train, X_test_norm, y_test_truth)
```

timing info:

stage	sample size	execution time (s)
training on train set	1000	0.193562
classifying test set	3836	0.119823
total	--	0.313385

ROC:

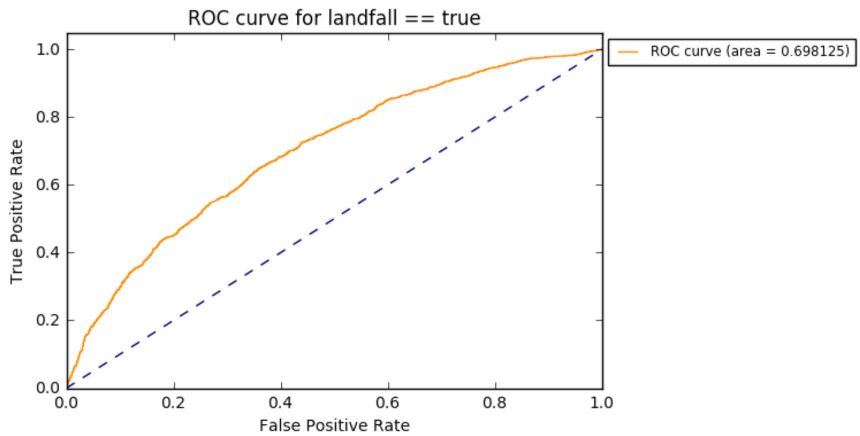
metric	score
AUC	0.698125

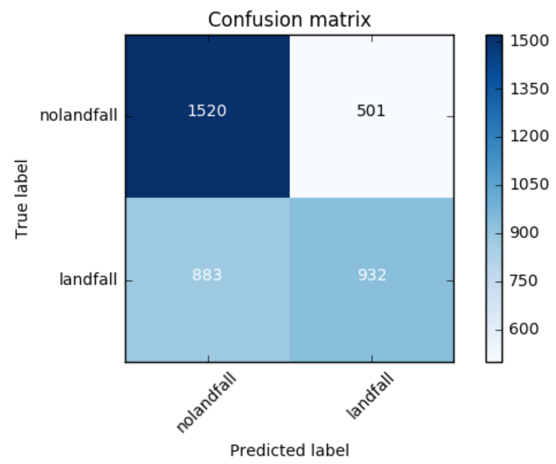
classification scores:

metric	score
accuracy_score	0.639208
average_precision_score	0.697035
f1_score	0.573892
recall_score	0.513499

classification report:

```
             precision    recall  f1-score   support

 nolandfall       0.63      0.75      0.69      2021
   landfall       0.65      0.51      0.57      1815

avg / total       0.64      0.64      0.63      3836
```





In [350]:

```
# train and classify on standardized datasets
train_and_classify(clf, X_train_std, y_train, X_test_std, y_test_truth)
```

timing info:

stage	sample size	execution time (s)
training on train set	1000	0.164902
classifying test set	3836	0.097948
total	--	0.262850

ROC:

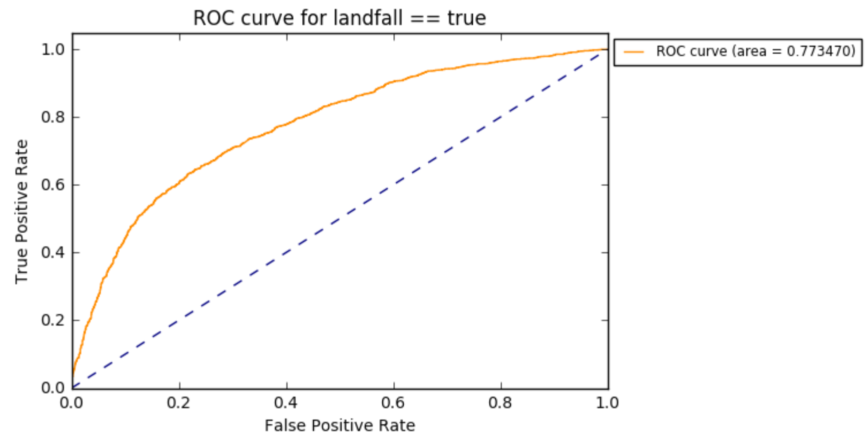
metric	score
AUC	0.773470

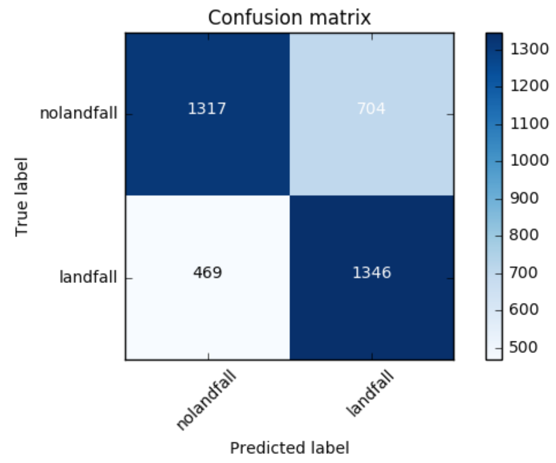
classification scores:

metric	score
accuracy_score	0.694213
average_precision_score	0.760223
f1_score	0.696507
recall_score	0.741598

classification report:

```
             precision    recall  f1-score   support

 nolandfall       0.74      0.65      0.69      2021
   landfall       0.66      0.74      0.70      1815

avg / total       0.70      0.69      0.69      3836
```





SVM Analysis

The 10-fold ShuffleSort cross-validation run of the SVM classifier showed that running it on the standardized training set yields the best performance in terms of AUC but not by much:

data	accuracy	mean AUC
raw	0.53 (+/- 0.09)	0.635947
normalized	0.64 (+/- 0.05)	0.650702
standardized	0.62 (+/- 0.04)	0.654708

Of note, the ROC curve for the SVM classifier's prediction on the raw test set shows that the slope of the curve rises quickly but immediately levels off. The ROC curves for the predictions on the normalized and standardized test sets don't exhibit this behavior and thus we see that for our tropical storm dataset, data rescaling helps the performance of the SVM classifier.

The following table aggregates the accuracy and AUC scores from the above classification runs on the test set:

data	accuracy	AUC
raw	0.632690	0.678275
normalized	0.639208	0.698125
standardized	0.694213	0.773470

Here the choice is clear however the results don't resemble what we saw in the cross-validation of the training set. The best performance in terms of accuracy and AUC comes from running the SVM classifier on the standardized training and test data sets.

Final Analysis

To summarize, I took the IBTrACS tropical storm dataset and performed ETL tasks to clean up missing values from the dataset and separate out the class variable and features. I derived the class variable **landfall** by iterating over the value of the **landfall** variable for each storm's track point and checking if the value ever equaled 0 (made landfall). If so, then the class variable **landfall** was set to True. Otherwise, the storm did not reach land and the class variable **landfall** was set to False.

class label distribution of filtered source dataset:

- total storms: 4836
- total storms with class variable landfall == True: 2315
- total storms with class variable landfall == False: 2521

From the 2315 storms that made landfall, I randomly sampled 500 instances to include in the raw training set. Similarly, from the 2521 storms that did not make landfall, I randomly sampled 500 instances to contribute to the raw training set as well. In total, 1000 storms were randomly sampled from their respective class label and separated out as the raw training set. The remainder of the storms (3836) were separated out as the raw test set.

breakdown of dataset records used for training and test sets:

- total storms: 4836
- total storms used for training set : 1000
- total storms used for test set: 3836

training set class label distribution of randomly sampled datasets of each class value:

- total training set storms: 1000
- total training set storms with class variable landfall == True : 500
- total training set storms with class variable landfall == False: 500
- total test set storms: 3836

To test how scaling our raw training and test set could help the machine learning algorithms perform better, I also created alternate versions of the dataset using 2 approaches. First I took the raw training and test set and normalized them so that all the feature values rescaled to a value in the range of 0 to 1. This is our normalized version of training and test set. Secondly, I took the raw training and test set and standardized them so that the distribution of each feature was shifted to have a mean of 0 and a standard deviation of 1. This is our standardized version of the training and test set.

dataset preparations

- raw training and test (no rescaling)
- normalized training and test (rescaled to the range of 0 and 1)
- standardized training and test (shifted so that the mean is 0 and standard deviation is 1)

As for the machine learning algorithms, I decided to use the 5 algorithms that we went over in our course:

- Decision Tree
- Naïve Bayes
- Logistic Regression

- Nearest Neighbor
- Support Vector Machines

In determining how well an algorithm performs on our dataset, I came to the conclusion that although the accuracy score is important, the cost associated with predicting false negatives (those storms that our algorithms predicted would not make landfall but actually did make landfall) was far too great, with implications on the impact on human lives and property. As such, the more important score is the AUC (area under the curve) of the ROC (receiver operating characteristics) curve.

The methodology I employed for assessing the performance of each algorithm was as follows:

1. For each of the 5 algorithms, take the raw training set and perform ShuffleSplit cross-validation 10 times to get the average accuracy and AUC scores.
2. Tweak the algorithm-specific parameters to explore settings that help increase the performance of the algorithm.
3. After settling on the settings, run the classifier on the raw test set and get the accuracy and AUC scores.
4. Repeat steps 1-3 for the normalized training and test set.
5. Repeat steps 1-3 for the standardized training and test set.
6. For each algorithm, select the data-specific (raw, normalized, or standardized) run on the test set that results in the highest AUC score first and accuracy score second.

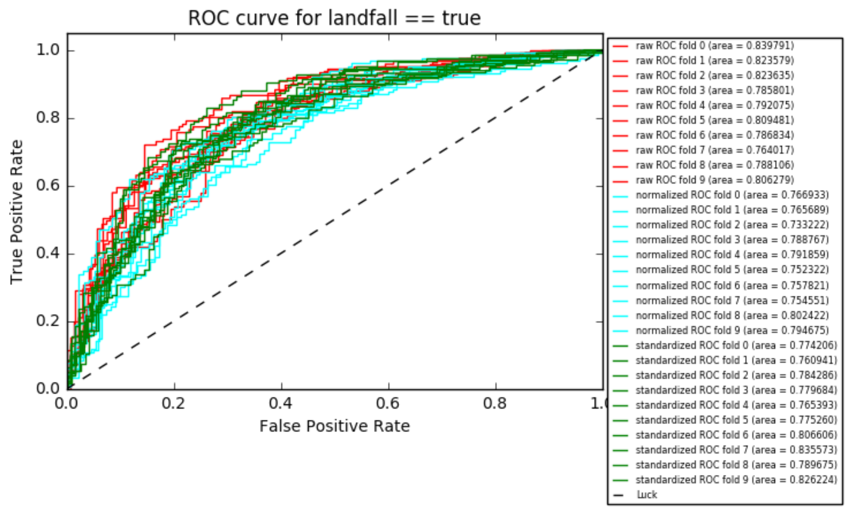
The results are as follows:

algorithm	data	accuracy	AUC
Decision Tree	raw	0.700209	0.760803
Naïve Bayes	standardized	0.642336	0.693288
Logistic Regression	raw	0.692127	0.752970
Nearest Neighbor	standardized	0.701512	0.763377
SVM	standardized	0.694213	0.773470

The top performer in terms of AUC score is the SVM algorithm run on the standardized data set. A close second but with a higher accuracy score is the Nearest Neighbor algorithm run on the standardized data set. The Decision Tree classifier run on the raw dataset also gives comparable performance scores. So how do I choose which classifier to use? Why choose? We can use a majority vote classifier that will essentially take any number of classifier models and classify a test set based on the majority vote. In our case, we can take our Decision Tree, Nearest Neighbor and SVM classifiers, register them in scikit-learn's VotingClassifier(), and run it on our test set. The issue here is that the VotingClassifier has to take in a single dataset type. In our case, the best Decision Tree results came from a run on the raw training and test set while the Nearest Neighbor and SVM classifiers got their best scores from running on the standardized training and test set. Comparing the accuracy and AUC scores of the entropy-based Decision Tree runs on the raw and standardized data sets shows a difference of .004 in terms of accuracy and .008 in terms of AUC. For our case, we should be fine with including the Decision Tree classifier into the VotingClassifier and feeding it the standardized data sets.

In [375]:

```
from sklearn import ensemble

# decision tree classifier using entropy for information gain
clf_dt = tree.DecisionTreeClassifier(criterion="entropy", max_depth=5)

# nearest neighbor classifier
n_neighbors = 15
clf_nn = neighbors.KNeighborsClassifier(n_neighbors, weights="distance")

# SVM classifier
clf_svm = svm.SVC(probability=True)

# voting classifier
clf = ensemble.VotingClassifier([('dt', clf_dt), ('nn', clf_nn), ('svm', clf_svm)],
                                voting='soft')

# cross validation
cross_validation_metrics(clf, trainset)
```



cross-validation scores (10 folds):

data	accuracy	mean AUC
raw	0.73 (+/- 0.06)	0.801649
normalized	0.74 (+/- 0.05)	0.770950
standardized	0.71 (+/- 0.06)	0.789771

In [376]:

```
# train and classify on standardized datasets
train_and_classify(clf, X_train_std, y_train, X_test_std, y_test_truth)
```

timing info:

stage	sample size	execution time (s)
training on train set	1000	0.172354
classifying test set	3836	0.358572
total	--	0.530926

ROC:

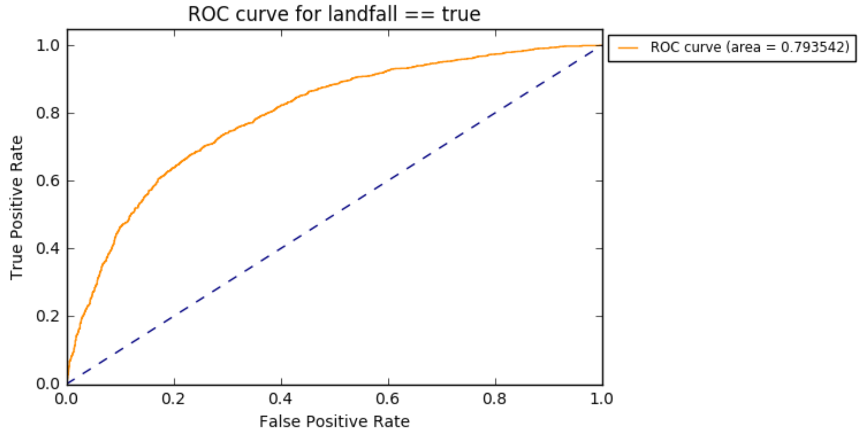
metric	score
AUC	0.793542

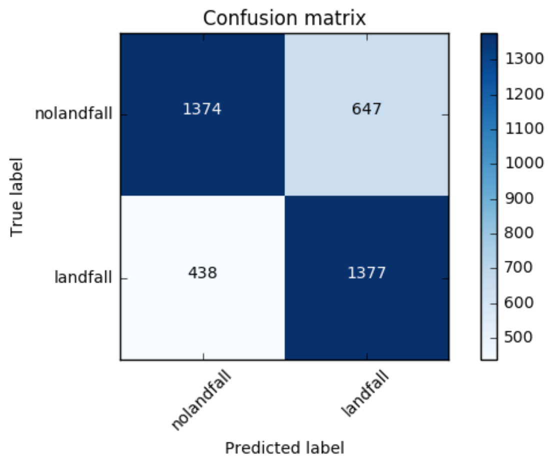
classification scores:

metric	score
accuracy_score	0.717153
average_precision_score	0.776598
f1_score	0.717374
recall_score	0.758678

classification report :

```
             precision    recall  f1-score   support

 nolandfall       0.76      0.68      0.72      2021
   landfall       0.68      0.76      0.72      1815

avg / total       0.72      0.72      0.72      3836
```

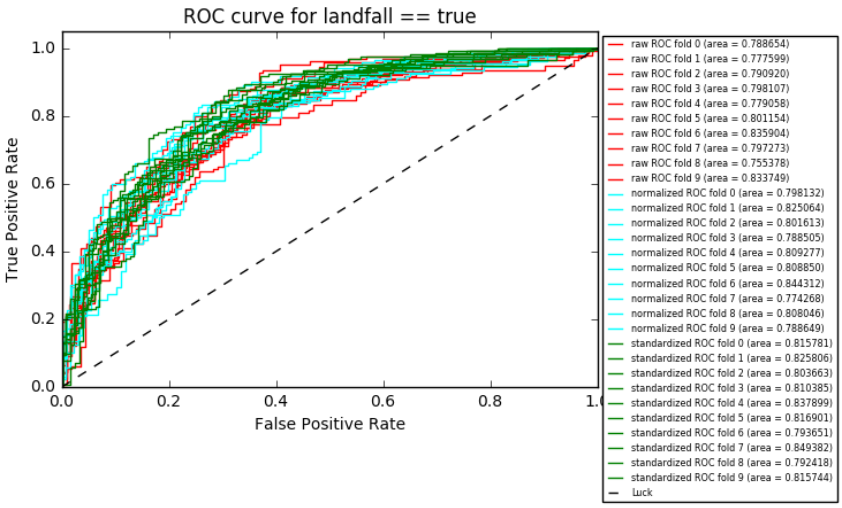




The accuracy and AUC scores of the VotingClassifier improves our scores and we nearly break the 80% AUC score. Can we improve the performance of the VotingClassifier by adding the rest of our learning algorithms?

In [377]:

```
# Bernoulli Naive Bayes classifier for data that is distributed according to multivariate Bernoulli distributions
clf_nb = naive_bayes.BernoulliNB()

# logistic regression classifier
clf_lr = linear_model.LogisticRegression()

# voting classifier
clf = ensemble.VotingClassifier([('dt', clf_dt), ('nn', clf_nn), ('svm', clf_svm),
                                 ('nb', clf_nb), ('lr', clf_lr)], voting='soft')

# cross validation
cross_validation_metrics(clf, trainset)
```



cross-validation scores (10 folds):

data	accuracy	mean AUC
raw	0.75 (+/- 0.05)	0.795673
normalized	0.73 (+/- 0.04)	0.804472
standardized	0.72 (+/- 0.04)	0.815915

In [378]:

```
# train and classify on standardized datasets
train_and_classify(clf, X_train_std, y_train, X_test_std, y_test_truth)
```

timing info:

stage	sample size	execution time (s)
training on train set	1000	0.178959
classifying test set	3836	0.423782
total	--	0.602741

ROC:

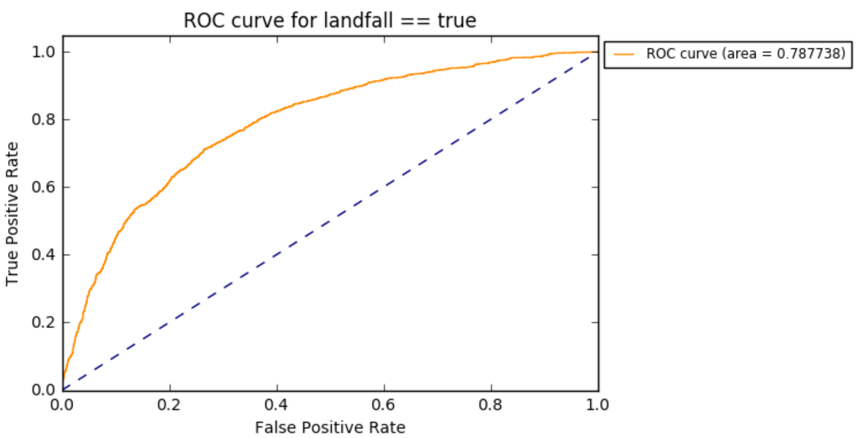
metric	score
AUC	0.787738

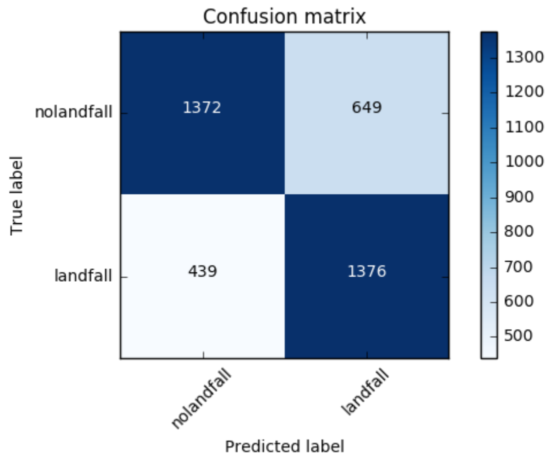
classification scores:

metric	score
accuracy_score	0.716371
average_precision_score	0.776038
f1_score	0.716667
recall_score	0.758127

classification report :

```
             precision    recall  f1-score   support

 nolandfall       0.76      0.68      0.72      2021
   landfall       0.68      0.76      0.72      1815

avg / total       0.72      0.72      0.72      3836
```





Adding in the Naive Bayes and Logistic Regression classifiers to the VotingClassifier did not improve the AUC or accuracy scores so we can leave them out as they did not add add any boost in performance.

The final results are shown:

algorithm	data	accuracy	AUC
Decision Tree	raw	0.700209	0.760803
Naïve Bayes	standardized	0.642336	0.693288
Logistic Regression	raw	0.692127	0.752970
Nearest Neighbor	standardized	0.701512	0.763377
SVM	standardized	0.694213	0.773470
Voting (DT+NN+SVM)	standardized	0.717153	0.793542