In [1]:
```
version = "REPLACE_PACKAGE_VERSION"
```

Assignment 4 - Tree-based classification; Synthesis Project

This assignment has two parts: the first part reviews tree-based classification, and the second part is a synthesis project.

In [2]:
```
# import some necessary libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

from sklearn import tree
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import classification_report

# Suppress all warnings
import warnings
warnings.filterwarnings('ignore')
```

Part 1: Classification. Predicting landscape types from satellite data.

Machine learning is being used increasingly to help fight climate change and promote sustainable land use. For this section of the assignment we will be working with a dataset derived from geospatial data, and our classification goal will be to predict the type of landscape visible by satellite. With an accurate classifier, we could monitor changes in land use and characteristics over time on a global scale.

Summary of the dataset: "The dataset is derived from two sources: 1) Landsat time-series satellite imagery from the years 2014-2015, and 2) crowdsourced georeferenced polygons with land cover labels obtained from OpenStreetMap. The crowdsourced polygons cover only a small part of the image area, and are used to extract training data from the image for classifying the rest of the image. The main challenge with the dataset is that both the imagery and the crowdsourced data contain noise (due to cloud cover in the images and inaccurate labeling/digitizing of polygons).

The 'landsat_training.csv' file contains the training data for classification. Do not use this file to evaluate classification accuracy because it contains noise (many class labeling errors).

The 'landsat_testing.csv' file contains testing data to evaluate the classification accuracy. Do not use this file to train the classifier. This file does not contain any class labeling errors.

Attribute Information (columns):

`class` : The land cover class (impervious, farm, forest, grass, orchard, water) **Note: this is the target variable to classify.**

`max_ndvi` : the maximum NDVI (normalized difference vegetation index) value derived from the time-series of satellite images.

`20150720_N - 20140101_N` : NDVI values extracted from satellite images acquired between January 2014 and July 2015, in reverse chronological order (dates given in the format yyyymmdd)."

This is a multi-class classification problem with six classes. Thus, you will want to examine both micro- and macro-averaged evaluation scores across all six classes, e.g. to see if there are landscape types that are more difficult than others to classify correctly.

Use the following code to read in the data files. Note that we convert the strings giving the landscape types (the "class" column to be predicted) to numeric labels using `LabelEncoder`.

In [3]:
```
df_train = pd.read_csv('assets/landsat_training.csv')
df_test  = pd.read_csv('assets/landsat_testing.csv')

from sklearn.preprocessing import LabelEncoder

enc = LabelEncoder()
df_train['class_code'] = enc.fit_transform(df_train['class'])
df_test['class_code']  = enc.transform(df_test['class'])

class_names = enc.classes_
print(class_names)

X_train = df_train.iloc[:,1:28]
y_train = df_train['class_code']

X_test = df_test.iloc[:,1:28]
y_test = df_test['class_code']
```

```
['farm' 'forest' 'grass' 'impervious' 'orchard' 'water']
```

Question 1. Basic decision trees (10 points)

Using `X_train` and `y_train` from the preceeding cell, train a `DecisionTreeClassifier` with default parameters and `random_state=0`. What are the 5 most important features found by the decision tree?

Your function should return a tuple of length 5 of the feature names in descending order of importance.

In [4]:
```
def answer_one():
    from sklearn.tree import DecisionTreeClassifier
    tree_clf = DecisionTreeClassifier(random_state=0).fit(X_train, y_tra
    feature_names = []
    "(" + str.strip('[]') + ")"
    for index, importance in enumerate(tree_clf.feature_importances_):
        feature_names.append([importance, X_train.columns[index]])

    feature_names.sort(reverse=True)
    feature_names = np.array(feature_names)
    feature_names = feature_names[:5,1]
    feature_names = feature_names.tolist()

    return tuple(feature_names)

answer_one()


    # YOUR CODE HERE
    #raise NotImplementedError()
```

Out[4]:
```
('max_ndvi', '20150125_N', '20141016_N', '20140813_N', '20140930_N')
```

In [5]:
```
# Autograder tests

stu_ans = answer_one()

assert isinstance(stu_ans, tuple), "Q1: Your function should return a tu
assert len(stu_ans) == 5, "Q1: The length of your returned tuple should
assert all([isinstance(item, str) for item in stu_ans]), "Q1: Your tuple


del stu_ans
```

Question 2. (10 points)

One very important decision tree parameter that can have a significant effect on how much the data are overfit (or not) is the *maximum tree depth*. Increasing maximum tree depth may increase accuracy by allowing more complex decision rules involving more features, but also increases the potential for overfitting with deeper trees.

Examine the effect of maximum tree depth on training and test set accuracy for a basic decision tree on this dataset. Return a list of `(train_score, test_score)` tuples for each possible maximum tree depth from 1 to 10.

Use `random_state=0` for any calls involving randomization.

In [6]:
```
def answer_two():
    from sklearn.tree import DecisionTreeClassifier
    score_list=[]
    for i in range (1,11):
        tree_clf = DecisionTreeClassifier(random_state=0, max_depth=i).f
        s=tree_clf.score(X_train, y_train)
        s_test=tree_clf.score(X_test, y_test)
        score_list.append((s,s_test))
    #feature_names = []
    #for index, importance in enumerate(tree_clf.feature_importances_):
        #feature_names.append([importance, X_train.columns[index]])

    #feature_names.sort
    #feature_names = np.array(feature_names)
    #feature_names = feature_names[:10,1]
    #feature_names = feature_names.tolist()

    return score_list

answer_two()

    # YOUR CODE HERE
    #raise NotImplementedError()
```

Out[6]:
```
[(0.725936462778568, 0.3433333333333333),
 (0.7871028923660502, 0.43333333333333335),
 (0.8349928876244666, 0.49333333333333335),
 (0.8614509246088193, 0.5433333333333333),
 (0.8744428639165481, 0.48),
 (0.8924608819345662, 0.53),
 (0.9128496917970602, 0.5566666666666666),
 (0.9354196301564722, 0.5533333333333333),
 (0.9533428165007113, 0.5566666666666666),
 (0.9664295874822191, 0.55)]
```

In [7]:
```
# Autograder tests

stu_ans = answer_two()

assert isinstance(stu_ans, list), "Q2: Your function should return a lis
assert len(stu_ans) == 10, "Q2: The length of your returned list should
assert all([isinstance(item, tuple) for item in stu_ans]), "Q2: Your lis

del stu_ans
```

Uncommenting the code below, take a look at the classification report for this GBDT model using the *test* set to evaluate. It is instructive to compare these results with the results from previous classifiers.

In [8]:
```
# Remember to comment them out before submitting the notebook

# test_accs = [t for _, t in answer_two()]
# best_tree_model = tree.DecisionTreeClassifier(random_state=0, max_depth
# y_pred = best_tree_model.predict(X_test)
# print(classification_report(y_test, y_pred, target_names=class_names))
# del test_accs, best_tree_model, y_pred
```

Question 3. Gradient-boosted decision trees (10 points)

We're now going to apply GBDT to the same dataset to compare its performance with ordinary decision trees. Train a default GBDT and output the training and test scores (accuracy) in that order, as a float 2-tuple. (Use `random_state=0` for any randomized operations.)

In [9]:
```
def answer_three():
    from sklearn.ensemble import GradientBoostingClassifier
    tree_clf = GradientBoostingClassifier(random_state=0).fit(X_train, y_
    s=tree_clf.score(X_train, y_train)
    s_test=tree_clf.score(X_test, y_test)

    return (s, s_test)

answer_three()
```

Out[9]:
```
(0.9793266951161688, 0.6233333333333333)
```

In [10]:
```
#Autograder tests

stu_ans = answer_three()

assert isinstance(stu_ans, tuple), "Q3a: Your function should return a tu
assert len(stu_ans) == 2, "Q3a: The length of your returned tuple should
assert all([isinstance(item, float) for item in stu_ans]), "Q3a: Your tup

del stu_ans
```

Part 2: Synthesis project

Question 1. City of Detroit Prediction Problem (70 points)

The synthesis project is based on a data challenge from the Michigan Data Science Team (MDST (http://midas.umich.edu/mdst/)).

The Michigan Data Science Team (MDST (http://midas.umich.edu/mdst/)) and the Michigan Student Symposium for Interdisciplinary Statistical Sciences (MSSISS (https://sites.lsa.umich.edu/mssiss/)) have partnered with the City of Detroit to help solve one of the most pressing problems facing Detroit - building deterioration. Blight violations (https://detroitmi.gov/departments/department-appeals-and-hearings/blight-ticket-information) are issued by the city to individuals who allow their properties to remain in a deteriorated condition. Every year, the city of Detroit issues millions of dollars in fines to residents and every year, many of these fines remain unpaid. Enforcing unpaid blight fines is a costly and tedious process, so the city wants to know: how can we increase blight ticket compliance?

The first step in answering this question is understanding when and why a resident might fail to comply with a blight ticket. This is where predictive modeling comes in. For this assignment, your task is to predict whether a given blight ticket will be paid on time.

All data for this assignment has been provided to us through the Detroit Open Data Portal (https://data.detroitmi.gov/). **Only the data already included in your Coursera directory can be used for training the model for this assignment.** Nonetheless, we encourage you to look into data from other Detroit datasets to help inform feature creation and model selection. Some related datasets of interest include:

- Building Permits (https://data.detroitmi.gov/datasets/building-permits)
- Trades Permits (https://data.detroitmi.gov/datasets/trades-permits)
- Improve Detroit: Submitted Issues (https://data.detroitmi.gov/datasets/improve-detroit-issues)
- DPD: Citizen Complaints (https://data.detroitmi.gov/datasets/dpd-citizen-complaints)
- Parcel Data (https://data.detroitmi.gov/datasets/parcels-2)

We provide you with two data files for use in training and validating your models: train.csv and test.csv. Each row in these two files corresponds to a single blight ticket, and includes information about when, why, and to whom each ticket was issued. The target variable is compliance, which is True if the ticket was paid early, on time, or within one month of the hearing data, False if the ticket was paid after the hearing date or not at all, and Null if the violator was found not responsible. Compliance, as well as a handful of other variables that will not be available at test-time, are only included in train.csv.

Note: All tickets where the violators were found not responsible are not considered during evaluation. They are included in the training set as an additional source of data for visualization, and to enable unsupervised and semi-supervised approaches. However, they are not included in the test set.

File descriptions (Use only this data for training your model!)

```
train.csv - the training set (all tickets issued 2004-2011)
test.csv - the test set (all tickets issued 2012-2016)
addresses.csv & latlons.csv - mapping from ticket id to addresse
s, and from addresses to lat/lon coordinates.
 Note: misspelled addresses may be incorrectly geolocated.
```

Data fields

train.csv & test.csv

```
ticket_id - unique identifier for tickets
agency_name - Agency that issued the ticket
inspector_name - Name of inspector that issued the ticket
violator_name - Name of the person/organization that the ticket
was issued to
violation_street_number, violation_street_name, violation_zip_co
de - Address where the violation occurred
mailing_address_str_number, mailing_address_str_name, city, stat
e, zip_code, non_us_str_code, country - Mailing address of the v
iolator
ticket_issued_date - Date and time the ticket was issued
hearing_date - Date and time the violator's hearing was schedule
d
violation_code, violation_description - Type of violation
disposition - Judgment and judgement type
fine_amount - Violation fine amount, excluding fees
admin_fee - $20 fee assigned to responsible judgments
state_fee - $10 fee assigned to responsible judgments
late_fee - 10% fee assigned to responsible judgments
discount_amount - discount applied, if any
clean_up_cost - DPW clean-up or graffiti removal cost
judgment_amount - Sum of all fines and fees
grafitti_status - Flag for graffiti violations
```

train.csv only

```
payment_amount - Amount paid, if any
payment_date - Date payment was made, if it was received
payment_status - Current payment status as of Feb 1 2017
balance_due - Fines and fees still owed
collection_status - Flag for payments in collections
compliance [target variable for prediction]
 Null = Not responsible
 0 = Responsible, non-compliant
 1 = Responsible, compliant
compliance_detail - More information on why each ticket was mark
ed compliant or non-compliant
```

Evaluation

Your predictions will be given as the probability that the corresponding blight ticket will be paid on time.

The evaluation metric for this assignment is the Area Under the ROC Curve (AUC).

Your grade will be based on the AUC score computed for your classifier. A model which with an AUROC of 0.7 passes this assignment, over 0.75 will receive full points.

For this assignment, create a function that trains a model to predict blight ticket compliance in Detroit using `train.csv`. Using this model, return a series of length 61001 with the data being the probability that each corresponding ticket from `test.csv` will be paid, and the index being the ticket_id.

Example:

```
ticket_id
   284932    0.531842
   285362    0.401958
   285361    0.105928
   285338    0.018572
             ...
   376499    0.208567
   376500    0.818759
   369851    0.018528
   Name: compliance, dtype: float32
```

In [11]:
```
import pandas as pd
```

```
import numpy as np
from sklearn import preprocessing
from sklearn.linear_model import LogisticRegression

from sklearn.model_selection import train_test_split

def blight_model():

    train = pd.read_csv('assets/train.csv', encoding='ISO-8859-1')
    #test = pd.read_csv('assets/test.csv', encoding='ISO-8859-1')
    latlons = pd.read_csv('assets/latlons.csv')
    addresses = pd.read_csv('assets/addresses.csv')

    addresses_latlons = addresses.merge(latlons, how='inner', on='address

    train, test=train_test_split(train)
    print (list(train.columns))
    print(list(test.columns))


    for df in [train, test]:
        X = df.select_dtypes(include=['object'])
        le = preprocessing.LabelEncoder()
        X_2 = X.astype(str).apply(le.fit_transform)

        for col in X_2.columns:
            df[col] = X_2[col]

        df.fillna(method='pad', inplace=True)
        df.fillna(0, inplace=True)

    train_nonull = train[(train['compliance'] == 1) | (train['compliance
    y_train = train_nonull['compliance']

    X_train = train_nonull

    print(len(list(test.columns)))
    print(len(list(X_train.columns)))
    print (list(test.columns))
    print (list(X_train.columns))
    print (set (test.columns)-set(X_train.columns))
    print (set (X_train.columns)-set(test.columns))

    logit = LogisticRegression().fit(X_train, y_train)

    print(logit.score(X_train, y_train))

    y_score_logit = logit.decision_function(test)

    probs = logit.predict_proba(test)[:,1]
    test['prob'] = probs
```

```
    output = test.set_index('ticket_id')['prob']
    return output[:61001]
```

In [12]:
```
blight_model()
```

```
['Unnamed: 0', 'ticket_id', 'agency_name', 'inspector_name', 'violator
_name', 'violation_street_number', 'violation_street_name', 'violation
_zip_code', 'mailing_address_str_number', 'mailing_address_str_name',
'city', 'state', 'zip_code', 'non_us_str_code', 'country', 'ticket_iss
ued_date', 'hearing_date', 'violation_code', 'violation_description',
'disposition', 'fine_amount', 'admin_fee', 'state_fee', 'late_fee', 'd
iscount_amount', 'clean_up_cost', 'judgment_amount', 'payment_amount',
'balance_due', 'payment_date', 'payment_status', 'collection_status',
'grafitti_status', 'compliance_detail', 'compliance']
['Unnamed: 0', 'ticket_id', 'agency_name', 'inspector_name', 'violator
_name', 'violation_street_number', 'violation_street_name', 'violation
_zip_code', 'mailing_address_str_number', 'mailing_address_str_name',
'city', 'state', 'zip_code', 'non_us_str_code', 'country', 'ticket_iss
ued_date', 'hearing_date', 'violation_code', 'violation_description',
'disposition', 'fine_amount', 'admin_fee', 'state_fee', 'late_fee', 'd
iscount_amount', 'clean_up_cost', 'judgment_amount', 'payment_amount',
'balance_due', 'payment_date', 'payment_status', 'collection_status',
'grafitti_status', 'compliance_detail', 'compliance']
35
35
['Unnamed: 0', 'ticket_id', 'agency_name', 'inspector_name', 'violator
_name', 'violation_street_number', 'violation_street_name', 'violation
_zip_code', 'mailing_address_str_number', 'mailing_address_str_name',
'city', 'state', 'zip_code', 'non_us_str_code', 'country', 'ticket_iss
ued_date', 'hearing_date', 'violation_code', 'violation_description',
'disposition', 'fine_amount', 'admin_fee', 'state_fee', 'late_fee', 'd
iscount_amount', 'clean_up_cost', 'judgment_amount', 'payment_amount',
'balance_due', 'payment_date', 'payment_status', 'collection_status',
'grafitti_status', 'compliance_detail', 'compliance']
['Unnamed: 0', 'ticket_id', 'agency_name', 'inspector_name', 'violator
_name', 'violation_street_number', 'violation_street_name', 'violation
_zip_code', 'mailing_address_str_number', 'mailing_address_str_name',
'city', 'state', 'zip_code', 'non_us_str_code', 'country', 'ticket_iss
ued_date', 'hearing_date', 'violation_code', 'violation_description',
'disposition', 'fine_amount', 'admin_fee', 'state_fee', 'late_fee', 'd
iscount_amount', 'clean_up_cost', 'judgment_amount', 'payment_amount',
'balance_due', 'payment_date', 'payment_status', 'collection_status',
'grafitti_status', 'compliance_detail', 'compliance']
set()
set()
0.9262980146913903
```

Out[12]:
```
ticket_id
24319     0.409867
```

```
214560    0.006346
253442    0.005843
76702     0.012720
263952    0.005547
            ...
178794    0.059984
222124    0.002017
139785    0.003820
167671    0.006264
255704    0.015159
Name: prob, Length: 61001, dtype: float64
```

In [13]:
```
# Autograder tests - sanity checks; no points for passing this cell

stu_ans = blight_model()

assert isinstance(stu_ans, pd.Series), "Your function should return a pd
assert len(stu_ans) == 61001, "Your series is of incorrect length. "
```

```
['Unnamed: 0', 'ticket_id', 'agency_name', 'inspector_name', 'violator
_name', 'violation_street_number', 'violation_street_name', 'violation
_zip_code', 'mailing_address_str_number', 'mailing_address_str_name',
'city', 'state', 'zip_code', 'non_us_str_code', 'country', 'ticket_iss
ued_date', 'hearing_date', 'violation_code', 'violation_description',
'disposition', 'fine_amount', 'admin_fee', 'state_fee', 'late_fee', 'd
iscount_amount', 'clean_up_cost', 'judgment_amount', 'payment_amount',
'balance_due', 'payment_date', 'payment_status', 'collection_status',
'grafitti_status', 'compliance_detail', 'compliance']
['Unnamed: 0', 'ticket_id', 'agency_name', 'inspector_name', 'violator
_name', 'violation_street_number', 'violation_street_name', 'violation
_zip_code', 'mailing_address_str_number', 'mailing_address_str_name',
'city', 'state', 'zip_code', 'non_us_str_code', 'country', 'ticket_iss
ued_date', 'hearing_date', 'violation_code', 'violation_description',
'disposition', 'fine_amount', 'admin_fee', 'state_fee', 'late_fee', 'd
iscount_amount', 'clean_up_cost', 'judgment_amount', 'payment_amount',
'balance_due', 'payment_date', 'payment_status', 'collection_status',
'grafitti_status', 'compliance_detail', 'compliance']
35
35
['Unnamed: 0', 'ticket_id', 'agency_name', 'inspector_name', 'violator
_name', 'violation_street_number', 'violation_street_name', 'violation
_zip_code', 'mailing_address_str_number', 'mailing_address_str_name',
'city', 'state', 'zip_code', 'non_us_str_code', 'country', 'ticket_iss
ued_date', 'hearing_date', 'violation_code', 'violation_description',
'disposition', 'fine_amount', 'admin_fee', 'state_fee', 'late_fee', 'd
iscount_amount', 'clean_up_cost', 'judgment_amount', 'payment_amount',
'balance_due', 'payment_date', 'payment_status', 'collection_status',
'grafitti_status', 'compliance_detail', 'compliance']
['Unnamed: 0', 'ticket_id', 'agency_name', 'inspector_name', 'violator
_name', 'violation_street_number', 'violation_street_name', 'violation
_zip_code', 'mailing_address_str_number', 'mailing_address_str_name',
'city', 'state', 'zip_code', 'non_us_str_code', 'country', 'ticket_iss
ued_date', 'hearing_date', 'violation_code', 'violation_description',
'disposition', 'fine_amount', 'admin_fee', 'state_fee', 'late_fee', 'd
iscount_amount', 'clean_up_cost', 'judgment_amount', 'payment_amount',
'balance_due', 'payment_date', 'payment_status', 'collection_status',
'grafitti_status', 'compliance_detail', 'compliance']
set()
set()
0.9267934096490154
```

In [14]:
```
# Hidden autograder tests - whether AUC >= 0.55
```

In [15]:
```
# Hidden autograder tests - whether AUC >= 0.6
```

In [16]:
```
# Hidden autograder tests - whether AUC >= 0.65
```

In [17]:
```
# Hidden autograder tests - whether AUC >= 0.7
```

In [18]:
```
# Hidden autograder tests - whether AUC >= 0.75
```

In [ ]:
```

```