Project 3

Classification and Inference with Machine Learning

This notebook is arranged in cells. Texts are usually written in the markdown cells, and here you can use html tags (make it bold, italic, colored, etc). You can double click on this cell to see the formatting.

The ellipsis (...) are provided where you are expected to write your solution but feel free to change the template (not over much) in case this style is not to your taste.

Hit "Shift-Enter" on a code cell to evaluate it. Double click a Markdown cell to edit.

Write your partner's name here (if you have one).

Link Okpy

In [ ]:

```
from client.api.notebook import Notebook
ok = Notebook('project3.ok')
_ = ok.auth(inline = True)
```

Imports

In [ ]:

```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
from matplotlib.colors import LogNorm
import warnings
warnings.filterwarnings('ignore')
```

In this project, we will get acquainted with some of the well known machine learning techniques and use them for classification and regression. Specifically, we will use

- Linear models
- k-Nearest Neighbors
- Random Forests
- Support Vector Machines
- Gaussian Process
- Neural Networks

The performance of these algorithms does depend on 'hyperparameters' which need to be tuned to get optimal. This is primarily what we will investigate. Since a lot of these tunings are common to different algorithms, to avoid repition, we will not investigate all in each of them. That being said, still a lot of the knobs will be repeated and its recommended to write utility functions to make plots and minimize manual labor (copy-pasting).

Data

The data is provided in the file **"specz_data.txt"**. The columns of the file (length of 13) correspond to - spectroscopic redshift ('zspec'), RA, DEC, magnitudes in 5 bands - u, g, r, i, z (denoted as 'mu,' 'mg,' 'mr,' 'mi,' 'mz' respectively); Exponential and de Vaucouleurs model magnitude fits ('logExp' and 'logDev' http://www.sdss.org/dr12/algorithms/magnitudes/ (http://www.sdss.org/dr12/algorithms/magnitudes/)); zebra fit ('pz_zebra); Neural Network fit ('pz_NN') and its error estimate ('pz_NN_Err')

We will undertake 2 exercises -

- Regression
  - We will use the magnitude of object in different bands ('mu, mg, mr, mi, mz') and do a regression exercise to estimate the redshift of the object. Hence our feature space is 5.
  - The correct redshift is given by 'zspec', which is the spectroscopic redshift of the object. We will use this for training and testing purpose.

    Sidenote: Photometry vs. Spectroscopy

    The amount of energy we receive from celestial objects – in the form of radiation – is called the flux, and an astro- nomical technique of measuring the flux is photometry. Flux is usually measured over broad wavelength bands, and with the estimate of the distance to an object, it can infer the object's luminosity, temperature, size, etc. Usually light is passed through colored filters, and we measure the intensity of the filtered light.

    On the other hand, spectroscopy deals with the spectrum of the emitted light. This tells us what the object is made of, how it is moving, the pressure of the material in it, etc. Note that for faint objects making photometric observation is much easier.

    Photometric redshift (photoz) is an estimate of the distance to the object using photometry. Spectroscopic redshift observes the object's spectral lines and measures their shifts due to the Doppler effect to infer the distance.</i>
- *Classification*
  - *We will use the same magnitudes and now also the redshift of the object ('zspec') to classify the object as either Elleptical or Spiral. Hence our feature space is now 6.*
  - *The correct class is given by compring 'logExp' and 'logDev' which are the fits for Exponential and Devocular profiles. If logExp > logDev, its a spiral and vice-versa. We will use this for training and testing purpose. Since the classes are not explicitly given, generate a column for those (Classes can be ± 1. If it is 0, it does not belong to either of the class.)*

Prep 1. Cleaning

Read in the files to create the data (X and Y) for both regression and classification.
You will have to clean the data -

- *Drop the entries that are nan or infinite*
- *Drop the unrealistic numbers such as 999, -999; and magnitudes that are unrealistic. Since these are absolute magnitudes, they should be positive and high. Lets choose a magnitude limit of 15 as safe bet.*
- *For classification, drop the entries that do not belong to either of the class*

For regression, X and Y data is cleaned magnitudes (5 feature space) and spectroscopic redshifts respectively. For classification, X and Y data is cleaned magnitudes+spectroscopic redshifts respectively (6 feature space) and classees respectively.

In [ ]:

```
#Read in and create data

fname = 'specz_data.txt'
spec_dat=np.genfromtxt(fname,names=True)
print(spec_dat.dtype.fields.keys())

#convenience variable
zspec = spec_dat['zspec']
logExp, logDev = spec_dat['logExp'], spec_dat['logDev']
...

#Cleaning data for Regression

#Cleaning data for Classification
```

What is the size of your data (number of objects) before and after cleaning? (For both regression and classification)

In [ ]:

```
...
```

Prep 2. Visualization

The next step should be to visualize the data.
For regression

- *Make a histogram for the distribution of the data (spectroscopic redshift).*
- *Make 5 2D histograms of the distribution of the magnitude as function of redshift (Hint: https://matplotlib.org/devdocs/api/_as_gen/matplotlib.axes.Axes.hist2d.html (https://matplotlib.org/devdocs/api/_as_gen/matplotlib.axes.Axes.hist2d.html))*

For classification

- *Make 6 1-d histogram for the distribution of the data (6 features - zspec and 5 magnitudes) for both class 1 and -1 separately*

In [ ]:

```
#Redshift distribution of objects and colors
...
```

In [ ]:

```
#Redshift distribution of objects and colors based on the class
...
```

Prep 3. Preprocessing

- *Next, split the sample into training data and the testing data. We will be using the training data to train different algorithms and then compare the performance over the testing data. In this project, keep 80% data as training data and uses the remaining 20% data for testing.*
- *Often, the data can be ordered in a specific manner, hence shuffle the data prior to splitting it into training and testing samples.*
- *Many algorithms are also not scale invariant, and hence scale the data (different features to a uniform scale). All this comes under preprocessing the data. http://scikit-learn.org/stable/modules/preprocessing.html#preprocessing (http://scikit-learn.org/stable/modules/preprocessing.html#preprocessing)*
  Use StandardScaler from sklearn (or write your own routine) to center the data to 0 mean and 1 variance. Note that you only center the training data and then use its mean and variance to scale the testing data before using it.

  (Hint: How to get a scaled training data:
  Let the training data be: train = ("training X data", "training Y data")
  You can first define a StandardScaler: scale_xdata, scale_ydata = preprocessing.StandardScaler(), preprocessing.StandardScaler()
  Then, do the fit: scale_xdata.fit(train[0]), scaley.fit(train[1])
  Next, transform: scaled_train_data = (scale_xdata.fit_transform(train[0]), scale_ydata.fit_transform(train[1]))

- *For some algorithms, the size of this training data will be unweildy (for eg. for GP, we will have to use ake inverses of matrices). For this purpose, create a smaller training set of 2000 samples. Again, create another copy of this which is normalized to standard scale.*
- *Do this for both, classification and regression.*

In [ ]:

```
from sklearn import preprocessing
```

In [ ]:

```
def prepdata(...):
    '''Function to prepare train, test and validation data given X and Y data se
t given
    '''
    ...

...
```

In [ ]:

```
#Training and validation fraction
tf, vf = 0.8, 0

#Create the data (and subsized data) for Regression
...

#scale the data
scalex, scaley = preprocessing.StandardScaler(), preprocessing.StandardScaler()
...
```

In [ ]:

```
#Create the data (and subsized data) for Classification
...

#scale the data
scalexc = preprocessing.StandardScaler()
...
```

Prep 4. Metrics

The last remaining preperatory step is to write metric for gauging the performance of the algorithm. Write a function to calculate the 'RMS' error given (y_predict, y_truth) to gauge regression and another function to evaluate accuracy of classification.
In addition, for classification, we will also use confusion matrix

In [ ]:

```
from sklearn.metrics import confusion_matrix

def rms(y1, y2, ...):
    '''Calculate the RMS error given the truth and the prediction
    '''

def accuracy(y1, y2, ...):
    '''Calculate the accuracy given the truth and the prediction
    '''
```

1. Linear Regression

Try to fit a linear regression model to the data and answer the following questions -

- *What is the error (rms) for the training sample and the testing sample? Make a scatter plot of the truth against the predictions. (Though left unasked hereafter, you should do this for every algorithm and after doing any kind of regression)*
- *Does the answer change if you use preprocessed vs raw data? Should it?*
- *Look at the coefficients best fit by the linear model, Does the order of importance agree with your intuition based on the previous visualization of the data?*

(Hint:
Let "lin = LinearRegression()" (This is our model)
Also, let testN and trainingN be our test and training data (either scaled or unscaled). testN = ("test X data", "test Y data")
*First, do the fit using the training data: lin.fit(*trainN)*
Then, predict: predict = lin.predict(testN[0]) where testN[0] is test X data.)

In [ ]:

```
from sklearn.linear_model import LinearRegression
# http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LinearRegression.html
```

In [ ]:

```
#Linear Regression
print('For linear regression\n')

lin  = LinearRegression()

...
```

In [ ]:

```
print('Linear coefficients')
lin.coef_
```

Observation

...

Classification

Use logistic regression (http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegressionCV.html (http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegressionCV.html)) from linear model to perform classification and calculate the accuracy. (You can use LogisticRegressionCV().fit(...) and LogisticRegressionCV().predict(...)) Check the accuracy by measuring the same from confusion matrix as well. (http://scikit-learn.org/stable/modules/generated/sklearn.metrics.confusion_matrix.html (http://scikit-learn.org/stable/modules/generated/sklearn.metrics.confusion_matrix.html))

In [ ]:

```
from sklearn.linear_model import LogisticRegressionCV
```

In [ ]:

```
...

print('Confusion Matrix\n', ...)
print('Accuracy\n', ...)
```

2. Quadratic Regression

The simplest extension is fitting a polynomial model to the data where we take combinations of features to 'n'th order. Try to fit the quadratic model to the previous data and answer the same questions again.

Use the Pipeline and PolynomialFeatures from sklearn to create quadratic polynomial from the features.

In [ ]:

```
from sklearn.preprocessing import PolynomialFeatures
from sklearn.pipeline import Pipeline
# http://scikit-learn.org/stable/modules/generated/sklearn.pipeline.Pipeline.html
```

In [ ]:

```
#For Quadratic Regression

qmodel = Pipeline([('poly', PolynomialFeatures(degree=2, interaction_only=False)),
                   ('linear', LinearRegression(fit_intercept=False))])

...

#Scatter plot
```

In [ ]:

```
print('Quadratic coefficients')
qmodel.steps[1][1].coef_
```

Observation

...

Classification

Do a classification in a similar fashion and see if accuracy improves.

In [ ]:

```
#Classification
...
```

Hyperparamter methods

For the following sections algorithms, we will be varying hyperparameters to get the best model and build some intuition. There are various ways to do this and we will use 'Grid Search' methodology which simply tries all the combinations along with some cross-validation scheme. For most part, we will use 4-fold cross validation.
Sklearn provides GridSearchCV functionality for this purpose.

Do not overwrite these grid search-ed variables (and not only their result) since we will compare all the models together in the end

In [ ]:

```
from sklearn.model_selection import GridSearchCV, RandomizedSearchCV
# http://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html
# http://scikit-learn.org/stable/modules/generated/sklearn.model_selection.RandomizedSearchCV.html
```

3. k Nearest Neighbors

For regression, lets play with grid search using knn to tune hyperparmeters. Lets consider the following 3 hyperparameters -

- *Number of neighbors (vary this between 2-100, say)*
- *Weights of leaves (Uniform or Inverse Distance weighing)*
- *Distance metric (Eucledian or Manhattan distance - parameter 'p')*

Do a grid search on these parameters using 4 fold cross validation. Identify top 10 models and plot their mean scores, along the standard deviation.
Answer the following questions-

- *Is it always better to use more neighbors?*
- *Is it better to weigh the leaves, if yes, which distance metric performs better?*
- *For every parameter, make plots for the mean test score while marginalizing over other parameters. Which parameters seem to affect the performance most. and try to see which parameter is more important than others (we will do this for each and every method...so spend some time to see the format of output and write a function to do so)*
- *GridCV returns fitting and scoring time for every combination. You will find that scoring time is higher than training time. Why do you think is that the case?*

In [ ]:

```
from sklearn.neighbors import KNeighborsRegressor
# http://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsRegressor.html
```

Hint: (Read the documentations carefully for more detail.)

First, define the hyperparameters: parameters = {'n_neighbors':[2, 3, 5, 10, 15, 20, 25, 50, 100], 'weights': ['uniform', 'distance'], 'p':[1, 2]}

Specify the algorithm you want to use: e.g. knnr = KNeighborsRegressor()

Then, Do a grid search on these parameters using 4 fold cross validation: gcknn = GridSearchCV(knnr, parameters, cv=4)

*Do the fit: gcknn.fit(*train)*

(Let "train" be the training data where "train = ("train X data", "train Y data")"

Get results: `results = gcknn.cv_results_`

`cv_results_` *has the following dictionaries: "rank_test_score," "mean_test_score," "std_test_score," and "params" (See http://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html (http://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html))*

Then, you can identify top 10 models based on "rank_test_score" and print out their "params," along with their "mean_test_score" and "std_test_score". Plot their mean scores, along the standard deviation.

In [ ]:

```
#An example of parameters
parameters = {'n_neighbors':[2, 3, 5, 10, 15, 20, 25, 50, 100], 'weights':['unif
orm', 'distance'], 'p':[1, 2]}
knnr = KNeighborsRegressor()

...
```

Define some useful functions here

You might want to return here after the first algorithm to write some utility functions ad avoid copy pasting for finding the best models, creating plots etc. Declaration for 2 functions that might be useful are given, feel free to make more (or less)

Its recommended to spend some time to go through output format of GridSearchCV and write some utility functions to make the recurring plots for every parameter.
Grid Search returns a dictionary with self explanatory keys for the most part. Mostly, the keys correspond to (masked) numpy arrays of size = #(all possible combination of parameters). The value of individual parameter in every combination is given in arrays with keys starting from 'param_*' and this should help you to match the combination with the corresponding scores.
*For masked arrays, you can access the data values by using *.data*

(Hint:

Try this:

In [ ]:

```
for i, key in enumerate(parameters.keys()):
    order = results['param_%s'%key]
    print(key)
    print(order.data)
```

What does it print out? Think about how you can use this to make plots for the mean test score while marginalizing over other parameters)

In [ ]:

```
# This is only a suggestion. Do it as you like.
def topN(results, n=10, plot=True):
    '''Parse the result of CV and return top N results based on the score
    '''
    args = np.argsort(results['rank_test_score'])
    for i in range(n):
        ...

def plotparams(results, parameters):
    '''Parse the result of CV and plot the score by varying a single parameter
    '''
    ...
```

In [ ]:

```
# Identify top 10 models and plot their mean scores, along the standard deviatio
n.
topN(results)

# For each hyperparameter, make plots for the mean test score while marginalizin
g over other parameters
plotparams(results, parameters)
```

Plot timings for fitting and scoring

Hint: Assume that you got results from: `results = gcknn.cv_results_`

Then, get the scoring time: results['mean_score_time']

and the fitting time: results['mean_fit_time']

In [ ]:

```
# Time for fitting (For each hyperparameter, make plots for the mean_fit_time wh
ile marginalizing over other parameters))
...
```

In [ ]:

```
# Time for scoring (For each hyperparameter, make plots for the mean_score_time
 while marginalizing over other parameters))
...
```

Observations

... ...

In [ ]:

```
print('RMS error on the training data set is ')
```

Classification

In [ ]:

```
from sklearn.neighbors import KNeighborsClassifier
# http://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsC
lassifier.html
```

Here we will look at 4 different type of cross-validation schemes -

- *Kfold*
- *Stratified Kfold*
- *Shuffle Split*
- *Stratified Shuffle Split*

Do 4 different grid searches, one for each of these cross validation schemes, and identify top 3 models for each. remember to initiate each model with same random state
Answer the following questions-

- *Do the conclusions for any parameter from the regression case?*
- *Does the mean accuracy change?*
- *Does the variance in mean accuracy change?*

Give justification for these results

In [ ]:

```
from sklearn.model_selection import KFold, StratifiedKFold, ShuffleSplit, Strati
fiedShuffleSplit
```

In [ ]:

```
parameters = {'n_neighbors':[2, 3, 5, 10, 15, 20, 25, 50, 100], 'weights':['unif
orm', 'distance'], 'p':[1, 2]}
knnc = KNeighborsClassifier()

#Grid Search
gc = GridSearchCV(knnc, parameters, cv=KFold(4, random_state=100))
#Do the fit
...

gc2 = GridSearchCV(knnc, parameters, cv=StratifiedKFold(4, random_state = 100))
#Do the fit
...

gc3 = GridSearchCV(knnc, parameters, cv=ShuffleSplit(4, 0.1, random_state = 100
))
#Do the fit
...

gc4 = GridSearchCV(knnc, parameters, cv=StratifiedShuffleSplit(4, 0.1, random_st
ate = 100))
#Do the fit
...
```

Comparing different cross-validation schemes

In [ ]:

```
#Make plot for differet schemes (just as in regression)
...
```

Observation

...

In [ ]:

```
print('The accuracy for the testing data set is ')
print('For KFold\n',)
...
print('For Stratified shuffle split\n', )
...
```

In [ ]:

```
#Best model
...
```

Henceforth, which cross validation scheme should be used for classification?

4. Random Forests

In [ ]:

```
from sklearn.ensemble import RandomForestRegressor
# http://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForest
Regressor.html
```

The most important feature of the random forest is the number of trees in the ensemble. We will also play with the maximum depth of the trees.

- *Do a combined grid search on both these parameters and identify top 10 models.*
- *Are the scores of these models statistically different? Based on this, which architecture will you choose for your model?*
- *For every parameter, make the plot for fitting time. Based on this and the previous question, how many trees do you recommend keeping in the ensemble?*
- *Random forest also gives the importance of different parameters. See how this compares with the coefficients given by the linear model and your expectations*

In [ ]:

```
# Grid Search
# This will take few minutes
rf = RandomForestRegressor()
parameters = {'n_estimators':[10, 50, 150, 200, 300], 'max_depth':[10, 50, 100]}

gcrf = GridSearchCV(rf, parameters, cv=5)
# Do the fit
...
results = gcrf.cv_results_
```

In [ ]:

```
# Identify top 10 models and plot their mean scores, along the standard deviatio
n.
...

# For each hyperparameter, make plots for the mean test score while marginalizin
g over other parameters
...
```

Time scaling for different parameters

In [ ]:

```
# Time for fitting (For each hyperparameter, make plots for the mean_fit_time wh
ile marginalizing over other parameters))
...
```

In [ ]:

```
print('RMS error on the data is')
...
```

In [ ]:

```
print('Importance of feautres')
gcrf.best_estimator_.feature_importances_
```

Observation

Based on the above, we recommend using 100 trees

Classification

In [ ]:

```
from sklearn.ensemble import RandomForestClassifier
# http://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForest
Classifier.html
```

In [ ]:

```
#Grid search (This will take few minutes)

rfc = RandomForestClassifier()
parameters = {'n_estimators':[10, 50, 150, 200, 300], 'max_depth':[10, 50, 100]}

gcrfc = GridSearchCV(rfc, parameters, cv=StratifiedShuffleSplit(4, 0.1, random_s
tate = 100))

#Do the fit
...

results = gcrfc.cv_results_

# Identify top 10 models and plot their mean scores, along the standard deviatio
n.
...

# For each hyperparameter, make plots for the mean test score while marginalizin
g over other parameters
...
```

In [ ]:

```
print('The accuracy for the testing data set is ')
...
```

In [ ]:

```
print('Importance of feautres')
gcrfc.best_estimator_.feature_importances_
```

5. Support Vector Machines

In [ ]:

```
from sklearn.svm import SVR
# http://scikit-learn.org/stable/modules/generated/sklearn.svm.SVR.html
```

Since SVMs involve evaluating a kernel as well, which under current implementation of sklearn scales as atleast n^2 samples, we subsize our taining data to 2000 samples for grid search. Then we will evaluate the best model on the full training set

Further, since SVM are not scale invariant, hence use the scaled data.

The most important feature is kernel. We will try 3 kernels - linear, rbf and polynomial.

Since all three have different parameters, we will use different grid searches for all three with 3fold CV.

- *For polynomial kernel, use gamma, C, coef0 as parameters (this will be slow, so do not spam parameter space)*
- *For linear kernel, use epsilon, C as parameters*
- *For RBF kernel, use gamma, C as parameters.* ***Here, choose the values to be used on log-scale instead of linear scale****. This is one of the recommended practices for RBF kernel and SVM.*

It is instructive to read the documentation to see hwo different parameters affect behavior (and hence which are more important), and to first change a couple parameters manually to find the reasonable limits and the time taken in fitting.

For each, print the top 5 models and plot the scores marginalizing over parameters to identify the most import parameters

In [ ]:

```
parametersp = {"gamma":[0.1, 1], 'C':[0.1, 1, 2], 'coef0':[0, 1]}
parametersl = {'C':[0.1, 1, 2], "epsilon":[0.005, 0.01, 0.1, 0.5]}

C_range = np.logspace(-2, 2, 5)
gamma_range = np.logspace(-2, 2, 5)
parametersf = {"gamma":gamma_range, 'C':C_range}
```

In [ ]:

```
#Grid Search for polynomial kernel (This will take few minutes)
svrp = SVR(kernel='poly')
gcsvrp = GridSearchCV(svrp, parametersp, cv=3)
```

In [ ]:

```
#Grid Search for linear kernel (This will take few minutes)
...
```

In [ ]:

```
#Grid Search for RBF kernel (This will take few minutes)
...
```

In [ ]:

```
# (This will take few minutes)
# Do the fit
...

# Get the result for polynomial kernel
results1 = gcsvrp.cv_results_
# Get the result for linear kernel
...
# Get the result for RBF kernel
...

# Identify top 10 models and plot their mean scores, along the standard deviatio
n.
...

# For each hyperparameter, make plots for the mean test score while marginalizin
g over other parameters
...
```

In [ ]:

```
print('RMS error for best fit method of different kernels is ')
print('For polynomial, %0.3f'%)
print('For linear, %0.3f'%)
print('For rbf, %0.3f'%)
```

Make a 2-D heat map using pyplot.pcolor to see how the score changes with gamma and C. Do you see a trend in the values of gamma and C? Based on what you know about these parameters, does this make sense?

In [ ]:

```
def make2dkey(dt, k1, k2):
    k1 = 'param_%s'%k1
    k2 = 'param_%s'%k2
    l1 = np.unique(dt[k1])
    l2 = np.unique(dt[k2])
    w = np.zeros([l1.size, l2.size])
    for i, iv in enumerate(l1):
        for j, jv in enumerate(l2):
            w[i, j] = dt['mean_test_score'][(dt[k1] == iv) & (dt[k2] == jv)]
    return l1, l2, w

def plot2dkey(l1, l2, w, k1=False, k2=False):

    fig, ax = plt.subplots()
    im = ax.pcolor(w)
    plt.colorbar(im)
    ax.set_xticklabels(l2)
    ax.set_yticklabels(l1)
    ax.set_yticks(np.arange(w.shape[0]) + 0.5, minor=False)
    ax.set_xticks(np.arange(w.shape[1]) + 0.5, minor=False)
    ax.invert_yaxis()
    if k1:
        ax.set_ylabel(k1)
    if k2:
        ax.set_xlabel(k2)

    return fig, ax
```

In [ ]:

```
# Make a 2D heat map using the above routine
...
```

It is also instructive to do this exercise without normalizing the data. Do so for the polynomial and the rbf kernel. Report on the difference you find.
Again make the pcolor map between gamma and C. Has the trend changed, and is it in line with your expectations

In [ ]:

```
# Grid search
svrf2 = SVR(kernel='rbf')
gcsvrf2 = GridSearchCV(svrf2, parametersf, cv=3)

# Do the fit
...
# Get the result
results = gcsvrf2.cv_results_

# Identify top 10 models and plot their mean scores, along the standard deviatio
n.
...

# For each hyperparameter, make plots for the mean test score while marginalizin
g over other parameters
...
```

In [ ]:

```
# Make a 2D heat map
...
```

In [ ]:

```
print('RMS error for best fit method of different kernels is ')
print('For rbf kernel with normalized data, %0.4f')
print('For rbf kernel with unnormalized data, %0.4f')
```

Observations

...

Classification

Try this using class weights and without. What do you naively expect and what do you get?
For some reason putting weights seems to make the performance worse *Find the best fit values of other parameters*

In [ ]:

```
from sklearn.svm import SVC
# http://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html
```

In [ ]:

```
# Grid search and plot
svc = SVC()
parameters = {'C':[0.1, 1, 2], 'gamma':[0.1, 1]}
C_range = np.logspace(-2, 2, 5)
gamma_range = np.logspace(-2, 2, 5)
parameters = {"gamma":gamma_range, 'C':C_range}

gsvcn = GridSearchCV(svc, parameters, cv=4)
# Do the fit
...
```

6. Gaussian Process

GPs are another kernel method of regression and classification. Here, since we need to invert the kernel matrix which is an N^3 process, it is not possible for us to work with full data as such. Hence we need methods to lower the rank of the kernel.
The most trivial way is to use less training data. We will do this and used reduced training data for the sklearn algorithm.

Another rank reduction technique is to decompose kernel matrix. This is based on the section 8.1 of http://www.gaussianprocess.org/gpml/chapters/RW8.pdf (http://www.gaussianprocess.org/gpml/chapters/RW8.pdf)
However since this process can lead to numerical instabilities, we will follow the algorithm in https://pubs.giss.nasa.gov/docs/2009/2009_Foster_fo04000r.pdf (https://pubs.giss.nasa.gov/docs/2009/2009_Foster_fo04000r.pdf), the V method elaborated in secion 5.2. You will need to go through section 2 and 3 as well to develop notation.

Based on the above, write a class GPsub with fit, reduce and predict function to model this reduced rank GP. The structure (functions in the classe) should broadly be as follows -

- ***init** - take in the value of lower rank, regularizer (alpha), error(sigma) on values, kernel and its assoicated parameters from the user.*
- *fit function - create the kernel matrix given the X and Y data*
- *predict - predict the mean values for the new data set X*
- *reduce - implement the V method form the paper above*

You do not need to define kernels, you can instead use them from the inbuilt GP class. We will try 2 kernels, polynomial (dot) and rbf.

Setup this class and make a scatter plot of the prediciton vs the truth for the training data set.

In [ ]:

```
from sklearn.gaussian_process import kernels
# http://scikit-learn.org/stable/modules/gaussian_process.html
```

In [ ]:

```
# This is only a suggestion
class GPsub:

    def __init__(self, sigma=2, alpha=1e-10, sub=100, l=1, kernel='poly'):

        rbf = kernels.RBF(length_scale=l)
        dot = kernels.DotProduct(sigma_0=sigma)
        if kernel is 'poly':
            self.ker = dot
        elif kernel is 'rbf':
            self.ker = rbf
        self.m = sub
        self.alpha = alpha

    def fit(self, X, Y):
        ...

    def reduce(self):
        ...

    def predict(self, Xt):
        ...
```

Grid Search

One this is set up, do the hyperparameter search for both the kernels. The parameters that need to be varied are

- *For rbf kernel, the rank to which the kernel is lowered and the length_scale*
- *For polynomial kernel, the rank to which the kernel is lowered and 'sigma'*

Make plot for the score on the test data set to judge the importance of every parameter.

- *How does increasing the rank of the parameter affect the score?*
- *What is the best length-scale. Is this in the ballpark of where you would naively expect it to be? Why or why not?*

In [ ]:

```
# Trial
gpsub = GPsub(l=20, kernel='rbf', sub=50)
# Let train = (train_xdata, train_ydata) be the training data
gpsub.fit(train[0], train[1])
# Let test = (test_xdata, test_ydata) be the test data
yym = gpsub.predict(test[0])
plt.plot(test[1], yym, 'b.')
plt.plot(test[1], test[1], 'k.')
plt.show()
print(rms(test[1], yym))

# Do a manual grid search for both the kernels and make an example scatter plot
...
```

In [ ]:

```
#Plot
```

In [ ]:

```
# Best model
gpsub = GPsub(l=..., kernel='rbf', sub=...)
# Do the fit
...
```

Inbuilt GP

Compare this with the GP from sklearn. Use the subsized training set with this model. Calculate the rms error and comment if our decomposition of the kernel helped us to improve in RMS error.

In [ ]:

```
from sklearn.gaussian_process import GaussianProcessRegressor
# http://scikit-learn.org/stable/modules/generated/sklearn.gaussian_process.Gaus
sianProcessRegressor.html
```

In [ ]:

```
kerdot = kernels.DotProduct(sigma_0=2)
gpdot = GaussianProcessRegressor(kernel=kerdot)
gpdot.fit(...)
zgpdot = scalesuby.inverse_transform(gpdot.predict(...))

kerrbf = kernels.RBF(length_scale=0.1)
gprbf = GaussianProcessRegressor(kernel=kerrbf, alpha=1e-1)
gprbf.fit(...)
zgprbf = (gprbf.predict(...))

# Make plot
...

# Calculate the rms
...
```

Comaprison

...

Classification

For this, feel free to extend the above GPsub class to include classification or simply use the inbuilt GPClassifier with subsized data

In [ ]:

```
from sklearn.gaussian_process import GaussianProcessClassifier
# http://scikit-learn.org/stable/modules/generated/sklearn.gaussian_process.Gaus
sianProcessClassifier.html
```

In [ ]:

```
# This will take 30-40 minutes!
kerrbf = kernels.RBF(length_scale=10)
gprbfc = GaussianProcessClassifier(kernel=kerrbf)
gprbfc.fit(...)
...
```

7. Neural Network

In [ ]:

```
from sklearn.neural_network import MLPRegressor
# http://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPRegressor.html
from sklearn.neural_network import MLPClassifier
# http://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPClassifier.html
```

For a neural network, the important hyperparameters are -

- *Number and size of layers*
- *Activation function*
- *Strength of regularization*
- *Batch size for training*
- *Training algorithm*

There are other parameters such as initial learning rate, parameters corresponding to the training algorithm etc., however we will not bother with these at the moment.

First, lets make a decision on the activation function and training algorithm since those are finite in numbers and then do the grid search on other parameters that are unconstrained

For neural network, we will only be using the normalized data set

Train a double layer network of size [30, 15] (arbitrary) using all the available activation functions and calculate the rms error. Based on this, decide on an activation function

In [ ]:

```
nnr = MLPRegressor([30, 10], max_iter = 1000)
parameters = {'activation':['relu', 'logistic', 'identity']}
gcnn = GridSearchCV(nnr, parameters, cv=4)

# Do the fit
...
# Get the result
results = gcnn.cv_results_

print(results['mean_test_score'])
```

Training Algorithm

Using the best activation function from above, try the 3 available algorithms - adam, SGD (without Nestrov Momentum), SGD (with Nestrov Momentum). Use starting learning rate = 0.001. Plot the loss_curve for all three algorithms, as well as see the wall clock time (use time.time package or %timeit functionality) for all three. To keep things consistent, start from same random state for all algorithms.
Which is the best algorithm?

In [ ]:

```
from time import time
```

In [ ]:

```
#train 3 networks and output time for them. Make a plot for loss_curve

lrate = [1e-3]
nnl = [[], [], []]

for lr in lrate:
    nnr1 = MLPRegressor([50, 30], max_iter = 1000, solver='adam', random_state=1
00, learning_rate_init=lr)
    %timeit nnr1.fit(...)
    nnl[0].append(nnr1)

    nnr2 = MLPRegressor([50, 30], max_iter = 1000, solver='sgd', random_state=10
0, learning_rate_init=lr)
    %timeit nnr2.fit(...)
    nnl[1].append(nnr2)

    nnr3 = MLPRegressor([50, 30], max_iter = 1000, solver='sgd', nesterovs_momen
tum=False, random_state=100, \
                        learning_rate_init=lr)
    %timeit nnr3.fit(...)
    nnl[2].append(nnr3)

# Make plot

i = 0
plt.plot(nnl[0][i].loss_curve_,'r', label='Adam')
plt.plot(nnl[1][i].loss_curve_,'b', label='SGD+nestrov')
plt.plot(nnl[2][i].loss_curve_, 'g', label='SGD')

plt.legend()
```

Batch Size

For different batch sizes, ranging from 10 to the size of training sample, plot the loss_curve as well as wall clock time. Again, remember to start from the same random state.
Explain the trend (roughly) seen in the loss curve and the wall clock time as a function of batch size

In [ ]:

```
batches = np.logspace(3, 8.2, 10, dtype=int, base=np.e)
batches

nnrb = []
times = []
for batch in batches:
    nnr1 = MLPRegressor([50, 30], max_iter = 1000, solver='adam', random_state=1
00, batch_size=int(batch), \
                        early_stopping=False)
    start = time()
    nnr1.fit(...)
    end = time()
    times.append(end - start)
    nnrb.append(nnr1)
```

In [ ]:

```
fig, ax = plt.subplots(1, 2, figsize = (12, 5))

...

plt.suptitle('Effect of batch size')
```

Observations

Upon increasing the batch size the number of iterations decreases because the weights are getting updated more often. The wall clock time however first decreases and then increases because now we are inverting biggere matrices, however

Grid Search

Do a grid search on different architecture of layers (number and sizes), batch sizes, and regularizing strength and find the top 10 models.

In [ ]:

```
# This will take few minutes.
nnr = MLPRegressor(max_iter = 1000)
parameters = {'hidden_layer_sizes':[5, 100, [5, 10], [100, 50]],
              'alpha':[1e-1, 1e-3, 1e-5],
             'batch_size':[50, 500, 2000]}

gcnn = GridSearchCV(nnr, parameters, cv=4)

# Do the fit
gcnn.fit(...)
results = gcnn.cv_results_

# Identify top 10 models and plot their mean scores, along the standard deviatio
n.
...

# For each hyperparameter, make plots for the mean test score while marginalizin
g over other parameters
...
```

In [ ]:

```
print('RMS error')
...
```

Classification

First, Lets confirm which activation function works the best for classification. Is the difference as significant as for the regression problem?

In [ ]:

```
mlpc = MLPClassifier([30, 10], max_iter = 1000)

parameters = {'activation':['relu', 'logistic', 'identity']}
gcnnc = GridSearchCV(mlpc, parameters, cv=4)

# Do the fit
gcnnc.fit(...)
results = gcnnc.cv_results_
print(results['mean_test_score'])
```

Grid Search

Do a grid search on hidden layers, batch and regularization to get the best model

In [ ]:

```
# This will take few minutes
mlpc = MLPClassifier(max_iter = 1000)

parameters = {'hidden_layer_sizes':[5, 100, [5, 10], [100, 50]],
              'alpha':[1e-1, 1e-3, 1e-5],
             'batch_size':[50, 500, 2000]}

gcnnc = GridSearchCV(mlpc, parameters, cv=4)
# Do the fit
gcnnc.fit(...)
results = gcnnc.cv_results_

# Identify top 10 models and plot their mean scores, along the standard deviatio
n.
...

# For each hyperparameter, make plots for the mean test score while marginalizin
g over other parameters
...
```

8. Compare!

- *Make a plot for the RMS error for regression using different algorithms on the testing data set using the best model from grid search*
- *Make a plot for the accuracy for classification using different algorithms on the testing data set using the best model from grid search*

In [ ]:

```
#Testing error for regression
...
```

In [ ]:

```
#Testing error for classification
...
```

To Submit

Execute the following cell to submit. If you make changes, execute the cell again to resubmit the final copy of the notebook, they do not get updated automatically.
We recommend that all the above cells should be executed (their output visible) in the notebook at the time of submission.
Only the final submission before the deadline will be graded.

In [ ]:

```
_ = ok.submit()
```