CHEAT SHEET PYTHON METRICS

ML studies how to automatically learn to make accurate predictions based on past observations.

Contents

- **Import packages**

```
# data modules
import numpy as np
import scipy.stats as stats
import pandas as pd

# For plotting
import matplotlib.pyplot as plt
import seaborn as sns
sns.set_style('whitegrid')

# TO make sure charts appear in the notebook:
%matplotlib inline
%config InlineBackend.figure_format ='retina'
```

- **Read data**

```
df_path = ('C:/Users/mjors_000/Desktop/ECON628-01-berenger92/datasets/affairs/affair.csv')
df=pd.read_csv(df_path)
```

- **For checking data**

```
df.head(3)

df.tail(2)
```

- **Info**

```
print df.shape

print df.info()
```

- **Describe**

```
df.describe()

df.describe().T
```

- **Sorting**

```
df.sort(['hhid','year'], ascending=True)
```

- **Locate some Rows**

```
# Just slicing some rows (400 - 405 )

df.iloc[400:405, :]
```

- **Matrix/heatmap**

For Table:

```
df.corr()
```

For Heatmap:

```
fig = plt.figure(figsize=(15,10))
ax = fig.gca()
sns.heatmap(df2.corr(), annot=True, linewidths=.15, cmap="YlGnBu", vmin=0, vmax=1, ax=ax)
plt.show()
```

- **Grouping/Tabulation tables**

 # average age of passengers group by Sex and survivorship

Method 1: (Best)

```
## Reset Index using as_index=False

titanic.groupby( ['Sex', 'Survived'], as_index=False)[ ['Age'] ].mean()
```

Method 2:

```
titanic.groupby( ['Sex', 'Survived'] )[ ['Age'] ].mean()
```

Method 3:

```
titanic['Age'].groupby( [ titanic['Sex'], titanic['Survived'] ] ).mean()
```

Method 4:

```
df.groupby(['subject_study'],
sort=True)[['low_salary_range']].mean().sort_values(by=['low_salary_range'],
ascending=False).reset_index()
```

- **Full Descriptive Statistics**

Method 1:

```
func_tuples = [('Mean', np.mean), ('Var', np.var), ('Std', np.std)]

titanic_sub_2 = titanic.groupby( ['Embarked','Survived', 'Sex'] )[ ['Age', 'Fare'] ].agg(func_tuples).reset_index()

titanic_sub_2
```

Method 2:

```
titanic.groupby(['Survived', 'Pclass'])[["Age"]].describe()

or

df[df<100].groupby('ID').agg(['mean','median','std'])
```

- **Boxplots**

```
fig = plt.figure(figsize=(4,3))
ax = fig.gca()
sns.boxplot(df.ln_farmsize, orient='v',
        fliersize=5, linewidth=3, notch=True,
```

```
        saturation=0.5, ax=ax)
ax.set_ylabel('Log farm size', fontsize=10)
ax.set_title('Boxplot Log Farm size', fontsize=13)
plt.show()
```

- **Histogram plotter**

```
histogram_plotter("Student's high salary", df.high_salary_range)

sns.distplot(lm_stats_1.resid, color='g')
```

- **Plotting Histograms by Group**

Method 1:

```
## Plot a kind='bar' of the number of Passengers by Pclass and Sex
titanic.groupby("Pclass").mean().plot(kind="bar", color="g", width=0.85)
plt.show()
```

Method 2:

```
## Plot a kind='bar' of the data set group by Pclass and Sex, use .size()
titanic.groupby(['Pclass','Sex']).size().plot(kind="bar", color='y')
plt.show()
```

- **Creating Log of yearly salary**

```
df['log_year_salary'] = df.yearly_salary.apply(np.log)
```

- **CLEAN DATA**
  - **Dropping variables/Row**

```
df.drop(['rainfall2', 'd_computer', 'dummy_negative_shock2'], inplace=True, axis=1)
df.drop("row_number", axis=0)
```

  - **Dropping Rows conditional on criteria**

```
df = df[df.line_race != 0]
or
df = df[df['line race'] != 0]
```

  - **Locating Missing Values of a column**

```
iowa.loc[iowa['Category'].isnull()]
```

  - **Replace Missing Data**

```
house_df.replace('?', np.nan, inplace=True)
house_df.ffill(inplace=True)
```

- **Replacing a number by NaN (missing value)**

iowa['Category'].replace(0, np.NaN)
or
df.applymap(lambda x: np.nan if x == '?' else x)

- **Dropping Column with any missing value**

df.dropna(axis=0, how=`any', inplace=True)

- **Dropping rows with any missing value**

df.dropna()

- **Dropping Duplicates**

df.groupby('id').apply(lambda x: x.drop('id', axis=1).drop_duplicates().shape[0]).reset_index()

- **Replacing Missing data with the Mean (of a certain group)**

df["pcexp"]=df.groupby('psu').transform(lambda m: m.fillna(m.mean()))

or
df.fillna(np.mean(df), inplace=True)

- **Sorting Dataset**

df.sort(['hhid','year'], ascending=True)

➢ **Choose the dataframe**

- **Choosing a subset dataframe**

#I'm defining a new subset dataframe, for my dependent variables.

df0 = pd.DataFrame(df, columns=['crop_residue', 'organic_fertil', 'modern_inputs_1'])

- **Merging Datasets**

Method 1:

 # left join: *it only merges observations that are only in both datasets*

pd.merge(df1, df2, on=0, how='left')

 # right join: *it only merges all observations that are in both datasets (keep everything)*
pd.merge(df1, df2, on=0, how='right')

 # outer join
pd.merge(df1, df2, on=0, how='outer')

 # inner join/same as left: *it only merges observations that are only in both datasets*
pd.merge(df1, df2, on=0, how='inner')

Method 2:

 # Define the two frames as one variable, frames.

```
frames = [df1, df2]
```

 # Concatenate

```
result = pd.concat(frames)
result
```

- **Functions**

```
def binary_affairs(x):
  if x==0:
    return 0
  else:
    return 1
```

- **MACHINE LEARNING - THE PROCESS**

•When solving problems using ML methods, the researcher needs to think of the larger data-driven problem-solving process of which these methods are a small part

1.Understand the problem and goal => Move from a vague premise to a concrete analytical formulation

2.Formulate it as a ML problem => classification or regression problem, build a model that detects anomalies as new data come in

3.Data exploration and preparation => EDA (exploratory data analysis), you need to carefully explore your data (missing values, need additional data)

4.Feature engineering => features = independent variables, predictors, factors, covariates. Feature transformation, interactions, aggregation of data over time and space

5.Method Selection => in ML you take a collection of methods and try them out empirically then validata which one works the best for your problem

6.Evaluation => Need to select the model that is the best (Model Evaluation process)

7.Deployment of model

- 2 major categories

1.**Supervised Learning methods** => problem when there is a target variable (continuos or discrete) that we want to predict or classify data into.

▪ Formally, this SLN methods predict a value of Y given input(s) X by learning (estimating, fitting or training) a function F. All the data points have labels

goal is to search for that function F that best predicts Y.

2.**Unsupervised Learning methods** => problem where there does not exist a target variable that we want to predict but we want to understand natural grouping patterns in the data. None of the data points have labels.

- **Machine Learning - Vocabulary**

•Learning => Use in the context of learning a model. Similar to fitting, or estimating a function, or training, or building a model

•Examples => This are the data points and instances.

•Features => Independent variables, attributes, predictor variables, and explanatory variables

•Labels => Response variable, dependent variable, and target variable

•Underfitting => Model is to simple and doesn't capture the structure of the data well enough

•Overfitting => Model is possibly too complex and models the noise in the data, which can result in poor generalization performance

•Regularization => General method to avoid overfitting by applying additional constraints to the model that is learned. Two common regularization are L 1

L1 = Lasso and L2 = Ridge

- **Machine Learning - Components**
- Evaluation Methodologies

•**Cross validation** Begins by splitting a label data set into k partitions (called folds), where k is typically set to 5 or 10. Cross validation then proceeds by iterating k times. In each iteration one of the k folds is held put as the test set, while the other k -1 folds are combined and used to train the model.

•Pros

▪ Every example is used in one test set for testing the model

▪ Each iteration gives us a performance estimate that can be aggregated/averaged to generate the overall estimate

- **Machine Learning - OLS - Assessing the accuracy of the model:**

Mean Squared Error (MSE)
R-Square

- **REGULARIZATION - ADDRESS THE ISSUE OF MULTICOLLINEARITY: RIDGE, LASSO AND ELASTICNET**

If you have 1000 variables in a dataset, you can't just do linear regression on all of them. Too much variables, and you will have multicollinearity.

2 ISSUES: UNDERFITTING, OVERFITTING.

machine learning will help overcoming the two issues by finding the perfect value of lambda.

#Lambda is a weight apply to the correction/penalty term of the Beta

GOAL: HERE WE ARE NOT TRYING TO EXPLAIN REG COEFFICIENTS, INSTEAD FINDING THE TECHNIQUE WITH THE SMALLEST MEAN SQUARE ERROR.

- **Why is normalization necessary for regularized regressions?**

ANS: Normalization is necessary for regularized regression because the beta values for each predictor variable must be on the same scale. If betas are different sizes just because of the scale of predictor variables the regularization term can't determine which betas are more/less important based on their size.

- **Command to Normalize X**

```
## Define y
y = df['TAX']
## Define X (excluding dependent variable)
columns_ = df.columns.tolist()
exclude_cols = ['TAX' ]
#TO EXCLUDE SOME VAR FROM THE WHOLE DATA FRAME
X = df[[i for i in columns_ if i not in exclude_cols]]
#IF THE VAR ARE NOT IN EXCLUDED LIST

## Print shapes of y and X
print y.shape, X.shape

# Initialize the StandardScaler object
from sklearn.preprocessing import StandardScaler
ss = StandardScaler()
# use the "fit_transform" function to normalize the X design matrix
Xn = ss.fit_transform(X)
```

- **Split Data into Train and Test**

Idea is to use only a sample of the dataset (30%), train the model on it, and test it on the left-out sample (70%).

  - **Command**

```
## import packages
from sklearn.cross_validation import cross_val_score
from sklearn.linear_model import LinearRegression
from sklearn.cross_validation import train_test_split
Xtrain, Xtest, ytrain, ytest = train_test_split(Xn, y, test_size=0.30, random_state=10)
print Xtrain.shape, Xtest.shape, ytrain.shape, ytest.shape
```

- **LINEAR REGRESSION**

```
## define a linear regression model
lr = LinearRegression()
```

```
## fit your model
lr.fit(Xtrain, ytrain)

## predict values:  you predict on the test, I am predict on the train so we can see how the model performs
ytrain_pred = lr.predict(Xtrain)
ytest_pred = lr.predict(Xtest)
# print(lr.coef_)
# print "====\n"

## Cross validate = 10
linreg_scores = cross_val_score(lr, Xtrain, ytrain, cv=10)

## Print the R^2
print linreg_scores
print "Average R^2 OLS score: %.3f" % (np.mean(linreg_scores))

## Use the R^2 and MSE to see how the model is performing on train and test data
from sklearn.metrics import r2_score
from sklearn.metrics import mean_squared_error

print('MSE - OLS train: %.3f, test: %.3f' % (
      mean_squared_error(ytrain, ytrain_pred),
      mean_squared_error(ytest, ytest_pred)))
print('R^2 0LS train: %.3f, test: %.3f' % (
      r2_score(ytrain, ytrain_pred),
      r2_score(ytest, ytest_pred)))
```

➢ RIDGE

The Ridge regression adds an additional thing to the loss function: the sum of the squared (non-intercept!) β values:

$$\text{minimize } RSS + Ridge = \sum_{i=1}^{n}\left(y_i - \left(\beta_0 + \sum_{j=1}^{p}\beta_j x_j\right)\right)^2 + \lambda_2 \sum_{j=1}^{p}\beta_j^2$$

•Hint: once the RidgeCV is fit, the attribute alpha_ contains the best alpha parameter it found through cross-validation.

•Ridge performs best searching alphas through logarithmic space (np.logspace).

Ridge assumes that x are related, shrink coefficients down, instead of deleting multicollinearity between Xs.

- **Commands for Ridge Regression**

```
from sklearn.linear_model import Ridge, Lasso, ElasticNet, RidgeCV, LassoCV, ElasticNetCV
ridge_alphas = np.logspace(0, 5, 200)
optimal_ridge = RidgeCV(alphas=ridge_alphas, cv=10)
#cv= cross validate to run on 10 different random samples

optimal_ridge.fit(Xtrain, ytrain)
print (optimal_ridge.alpha_)

# Cross-validate the Ridge $R^2$ with the optimal alpha.
ridge = Ridge(alpha=optimal_ridge.alpha_)
ridge_scores = cross_val_score(ridge, Xtrain, ytrain, cv=10)
print (ridge_scores)
print (np.mean(ridge_scores))

## Implement the Ridge Regression
ridge = Ridge(alpha=optimal_ridge.alpha_)

## Fit the Ridge regression
ridge.fit(Xtrain, ytrain)
ytrain_pred_ridge = ridge.predict(Xtrain)
ytest_pred_ridge = ridge.predict(Xtest)
# print(ridge.coef_)
# print "=====\n"

## Cross validate = 10
ridge_scores = cross_val_score(ridge, Xtrain, ytrain, cv=10)
## Print the R^2
print ridge_scores
print "Average R^2 Ridge score: %.3f" % (np.mean(ridge_scores))
#it prints average score of all R-square found in different test samples

print('MSE - Ridge train: %.3f, test: %.3f' % (
      mean_squared_error(ytrain, ytrain_pred_ridge),
      mean_squared_error(ytest, ytest_pred_ridge)))
print('R^2 - Ridge train: %.3f, test: %.3f' % (
      r2_score(ytrain, ytrain_pred_ridge),
      r2_score(ytest, ytest_pred_ridge)))
```

NB: We want MSE and R-square for train and test samples to be as close as possible

- **LASSO REGRESSION**

The Lasso regression takes a different approach. Instead of adding the sum of *squared* β coefficients to the RSS, it adds the sum of the *absolute value* of the β coefficients:

$$\text{minimize } RSS + Lasso = \sum_{i=1}^{n}\left(y_i - \left(\beta_0 + \sum_{j=1}^{p}\beta_j x_j\right)\right)^2 + \lambda_1 \sum_{j=1}^{p}|\beta_j|$$

$|\beta_j|$ is the absolute value of the β coefficient for variable x_j

•Hint: again, once the LassoCV is fit, the attribute alpha_ contains the best alpha parameter it found through cross-validation.

•Lasso, unlike Ridge, performs best searching alphas through linear space (np.linspace).

If you have 1000 variables in a dataset the Lasso can perform "feature selection" automatically for you.

Lasso does not reduce the value of correlated Xs, it just deletes one to remove multicollinearity (KEEP THE MOST RELEVANT ONES).

- **Command for Lasso Regression**

```
optimal_lasso = LassoCV(n_alphas=500, cv=10, verbose=1)
optimal_lasso.fit(Xtrain, ytrain)
print (optimal_lasso.alpha_)

# Cross-validate the Ridge $R^2$ with the optimal alpha.
lasso = Lasso(alpha=optimal_lasso.alpha_)
lasso_scores = cross_val_score(lasso, Xtrain, ytrain, cv=10)
print (lasso_scores)
print (np.mean(lasso_scores))

## Implement the Lasso Regression
lasso = Lasso(alpha=optimal_lasso.alpha_)

## fit your regression
lasso.fit(Xtrain, ytrain)
ytrain_pred_lasso = lasso.predict(Xtrain)
ytest_pred_lasso = lasso.predict(Xtest)
# print(lasso.coef_)
# print "====\n"

lasso_scores = cross_val_score(lasso, Xtrain, ytrain, cv=10)
print lasso_scores
print "Average R^2 Lasso score: %.3f" % (np.mean(lasso_scores))
print('MSE - Lasso train: %.3f, test: %.3f' % (
      mean_squared_error(ytrain, ytrain_pred_lasso),
      mean_squared_error(ytest, ytest_pred_lasso)))
print('R^2 - Lasso train: %.3f, test: %.3f' % (
      r2_score(ytrain, ytrain_pred_lasso),
      r2_score(ytest, ytest_pred_lasso)))
```

➢ ELASTIC NET REGRESSION

Elastic Net is a combination of both the Lasso and the Ridge regularizations. It adds both penalties to the loss function:

$$\text{minimize } RSS + Ridge + Lasso = \sum_{i=1}^{n}\left(y_i - \left(\beta_0 + \sum_{j=1}^{p}\beta_j x_j\right)\right)^2 + \lambda_1 \sum_{j=1}^{p}|\beta_j| + \lambda_2 \sum_{j=1}^{p}\beta_j^2$$

In the elastic net, the effect of the Ridge vs. the Lasso is balanced by the two lambda parameters.

Elastic Net combines Lasso and Ridge.

- **Command for Elastic Net Regression**

```
l1_ratios = np.linspace(0.01, 1.0, 50)
optim_elnet = ElasticNetCV(l1_ratio= l1_ratios, n_alphas=300, cv=10, verbose=1)
optim_elnet.fit(Xtrain, ytrain)
print (optim_elnet.alpha_)
print (optim_elnet.elastic_)

# Cross-validate the ElasticNet R2 with the optimal alpha and elastic.
•This is another way to look at the performance of your model.
•The tighter the distribution of residuals around zero, the better your model has performed!

enet = ElasticNet(alpha=optim_elnet.alpha_, l1_ratio =optim_elnet.l1_ratio_)
enet_scores = cross_val_score(enet, Xtrain, ytrain, cv=10)
print (enet_scores)
print (np.mean(enet_scores))


##  Create a model Enet
enet = ElasticNet(alpha=optim_elnet.alpha_, l1_ratio =optim_elnet.l1_ratio_)

## Fit your model
enet.fit(Xtrain, ytrain)

## predict
ytrain_pred_enet = enet.predict(Xtrain)
ytest_pred_enet = enet.predict(Xtest)
# print(enet.coef_)
# print "=====\n"

## Cross validate the scores and print the mean of the scores
enet_scores = cross_val_score(enet, Xtrain, ytrain, cv=10)
print enet_scores
print "Average R^2 Elastic Net score: %.3f" % (np.mean(enet_scores))
print('MSE – Elastic Net train: %.3f, test: %.3f' % (
      mean_squared_error(ytrain, ytrain_pred_enet),
      mean_squared_error(ytest, ytest_pred_enet)))
```

```
print('R^2 - ElasticNet train: %.3f, test: %.3f' % (
      r2_score(ytrain, ytrain_pred_enet),
      r2_score(ytest, ytest_pred_enet)))
```

- **Function to print the coefficients, their absolute values and the non-absolute values**

```
def best_reg_method(X, best_regulari):
   method_coefs = pd.DataFrame({'variable':X.columns,
                        'coef':best_regulari.coef_,
                        'abs_coef':np.abs(best_regulari.coef_)})
   method_coefs.sort_values('abs_coef', inplace=True, ascending=False)
   '''you can change the number inside head to display more or less variables'''
   return method_coefs.head(10)
```

- **Print out all Coefficients for the Method Chosen**

```
best_reg_method(X,lasso )
best_reg_method(X,enet )
```

- **Plot the residuals for the ridge, lasso, and elastic net on histograms**

```
## Fit
ridge.fit(Xtrain, ytrain)
enet.fit(Xtrain, ytrain)
lasso.fit(Xtrain,ytrain)
lr.fit(Xtrain,ytrain)

# residuals
ridge_resid = ytest - ridge.predict(Xtest)
lasso_resid = ytest - lasso.predict(Xtest)
enet_resid = ytest - enet.predict(Xtest)
lr_resid = ytest - lr.predict(Xtest)

#Plots
fig, axarr = plt.subplots(1, 3, figsize=(18, 6))
sns.distplot(lr_resid, bins=50, hist=True, kde=True,
          color='blue', ax=axarr[0], label='Linear Regression residuals')
sns.distplot(ridge_resid, bins=50, hist=True, kde=False,
          color='turquoise', ax=axarr[1], label='Ridge residuals')
sns.distplot(ridge_resid, bins=50, hist=True, kde=False,
         color='sienna', ax=axarr[2], label='Lasso residuals')
sns.distplot(ridge_resid, bins=50, hist=True, kde=False,
          color='orchid', ax=axarr[3], label='ElasticNet residuals')
plt.show()
```

- **Relation Mean-Variance and Coefficients**

How are the β coefficients affected by the mean and variance of your variables?

If the mean and variance of your x predictors are different, their respective β coefficients *scale with the mean and variance of the predictors* ***regardless of their explanatory power.***

➢ REGRESSION TREE (Classification and regression trees or CART models)

If we are interested to know not just general features, but specific predictors of Y (if male or if young)

* It involves stratifying or segmenting the predictor space into several simple regions.

* To make a prediction for a given observation, the method typically use the mean or the mode of the training observations in the region to which it belongs.

Idea: when the data has numerous features that interact in complicated nonlinear ways => creating a single linear global model can be remarkably challenging.

Terms

nodes represent the points on which a decision will be made

edges are the pathways between the answer and question nodes

To figure out which cell we are in, we start at the root node of the tree, and ask a sequence of questions about the features.

- **3 Models**

Cell Models

the model for l is the the sample mean of the dependent variable in that cell:

$$\hat{y} = \frac{1}{c}\sum_{i=1}^{c} y_i$$

* In regression trees we do this by ***maximizing*** I[C; Y] where Y is now the dependent variable, and C is the variable saying which leaf of the tree we end up at.

* But we can't do a direct maximization, so we need to do a greedy search => find the binary question that ***maximizes the information*** we get about Y, this will create the root node and 2 daughter's nodes. Then at each daughter node we repeat this procedure.

Tree Pruning

It ***selects a subtree*** that leads to a lower error rate CV (cross validation) and more.

Idea: In the minimization process we are more likely to create good predictions on the training set, but there is a high risk of overfitting the data => leads to a poor test performance.

Random Forest (Bagging => "Bootstrap Aggregating")

It trains M different trees on different subsets of the data, choosing randomly with replacement and then compute the ensemble. The data are chosen by selecting a random subset of features, and a random subset of observations to train model.

Random forests often have very good predictive accuracy, and reduces variance

- **COMMANDS FOR SIMPLE REGRESSION TREE**

Define y
y = data['log_inc']

Define X (exclude inc, incsq, log_inc)
columns_ = data.columns.tolist()
exclude_cols = ['inc', 'incsq', 'log_inc']
#TO EXCLUDE SOME VAR FROM THE WHOLE DATA FRAME

X = data[[i for i in columns_ if i not in exclude_cols]]
#IF THE VAR ARE NOT IN EXCLUDED LIST

Print shapes of y and X
print y.shape, X.shape

Train test split 70/30
from sklearn.cross_validation import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3)
Print shapes of X(s) and y(s)
print X_train.shape, y_train.shape
print X_test.shape, y_test.shape

Question: What is the benefit to splitting a dataset into some ratio of training and testing subsets for a learning algorithm?
Hint: What could go wrong with not having a way to test your model?

Answer: I can train the model using one part of the data and then evaluate it using the other part. That way I can figure out if I am overfitting to the training data or not by estimating model performance on unseen data. I can simulate what would happen if I applied it to new data in reality.

If I would not test my model, I could end up applying the model to new data, being very confident about my predictions, and then be very unhappy about the results.

#Build a regression tree
from sklearn.tree import DecisionTreeRegressor
dtr = DecisionTreeRegressor()

##Here is the gridsearch*, and we are asking the computer to go and find the one that gives the best model*

#First, we specify the parameters we want in a dictionary list with strings and values.

```
params = {"max_depth": [3,5,10,20],
        "max_features": [None, "auto"],
        "min_samples_leaf": [1, 3, 5, 7, 10],
        "min_samples_split": [2, 5, 7],
         "criterion": ['mse']
       }
```

Note 1: max_depth specifies how deep we want to search to go into the data. For us, the range is 3 - 20

Question: When the model is trained with a maximum depth of 1, does the model suffer from high bias or from high variance? How about when the model is trained with a maximum depth of 10? What visual cues in the graph justify your conclusions?
Hint: How do you know when a model is suffering from high bias or high variance?

Answer: ***• With a max_depth of 1, the model suffers from high bias.***
• With a max_depth of 10, the model suffers from high variance. The training score is much higher than the validation score. The 'max_depth' parameter can be thought of as how many questions the decision tree algorithm is allowed to ask about the data before making a prediction.

Other Answer: To ensure that you are producing an optimized model, you will train the model using the grid search technique to optimize the 'max_depth' parameter for the decision tree.

Note 2: Max features, defines the maximum number of independent variables we want to have.
n.b: none will overfit the model because we will have too much

Here crossvalidate using the Gridsearch

```
from sklearn.grid_search import GridSearchCV
dtr_gs = GridSearchCV(dtr, params, n_jobs=-1, cv=5, verbose=1)
```

#NB:njobs=-1 removes the criteria (one sample) after its already used.
#cv is cross validation and means the sample is split equally, trained, and then tested on 5 different samples.
#verbosity specifies the number of message the search will display. Higher verbosity means that as the search goes on, it prints more message about it.

Question - Gridsearch
What is the grid search technique and how it can be applied to optimize a learning algorithm?

Answer:
In general: grid search is a technique to find good values for model parameters or find an optimized model that cannot be optimized directly. It works by defining a grid over the model parameters, and then evaluating model performance for each point on the grid (using a validation set (or CV), not the training data). You can then choose the point on the grid that seems to work best. More specifically,

when we don't know which max depth is good for, us, 1 (underfitting=high bias) or 10 (overfitting=high variance), so we let the search deciding which parameters are the best for us.

Question - Cross-Validation

What is the k-fold cross-validation training technique?
What benefit does this technique provide for grid search when optimizing a model?
Hint: Much like the reasoning behind having a testing set, what could go wrong with using grid search without a cross-validated set?

Answer:
k-fold cross validation: For testing, we can split the data into one training set used to train a model and into another testing set used to evaluate model performance. E.g., this was the testing procedure used above (train_test_split). It is wasteful since we use one part of the data only for training and another part only for testing. We can do better by dividing the data into k folds, i.e., k equally large chunks of the entire dataset. We then iterate through the k chunks one by one and use the current chunk for model validation and the remaining k-1 chunks for training. What we end up with are k trained and evaluated models and we have used the entire dataset for validation. Averaging the validation scores gives us a single validation score. It is more reliable than if we had used only one split.

The benefits are that we can more reliably estimate model performance of various parameter configurations during grid search (less variance in the estimates). In grid search, there is a danger to overfit to the validation set since we use it many times to evaluate performance of different points on the grid and choose a point that delivered good performance. Hence, with more and more grid points, we are more and more likely to find a point that is good only by chance.
With cross validation, the overfitting problem is mitigated since our effective validation set size is larger. This comes at the cost of k-times the computational complexity of a single split.

#Fit the tree model : Now we need to bring everything together and build a model on the train data

```
dtr_gs.fit(X_train, y_train)
```

#Print best estimator, best parameters, and best score (best fit to explain Y)
''' dtr_best = is the regression tree regressor with best parameters/estimators'''

```
dtr_best = dtr_gs.best_estimator_
print "best estimator", dtr_best
print "\n==========\n"
print "best parameters",  dtr_gs.best_params_
print "\n==========\n"
print "best score", dtr_gs.best_score_
```

 # Produce the value for 'max_depth'

```
print("Parameter 'max_depth' is {} for the optimal model.".format(reg.get_params()['max_depth']))
```

Define Function to Print Feature importances

''' Here I am defining a function to print feature importance using best models'''

```
def feature_importance(X, best_model):
  feature_importance = pd.DataFrame({'feature':X.columns,
'importance':best_model.feature_importances_})
  feature_importance.sort_values('importance', ascending=False, inplace=True)
  return feature_importance
```

#Using the function

```
feature_importance(X, dtr_best)
```

#Predict on the Test Data

```
y_pred_dtr= dtr_best.predict(X_test)
y_pred_dtr
```

Evaluate the performance of your model (MSE in train and test data, R2 in train and test data)

```
from sklearn.metrics import r2_score
from sklearn.metrics import mean_squared_error
```

#Define Function that calls the MSE and R^2 at once, using the name of the method and calling the best model

```
def rsquare_meansquare_error(train_y, test_y, train_X, test_X, test, best_model):
  """ first we need to predict on the test and train data"""
  y_train_pred = best_model.predict(train_X)
  y_test_pred = best_model.predict(test_X)

  """ We call the MSE in the following lines"""
  print ('MSE ' + test + ' train data: %.2f, test data: %.2f' % (
    mean_squared_error(train_y, y_train_pred),
    mean_squared_error(test_y, y_test_pred)))

  """ We call the R^2 in the following lines"""
  print('R^2 ' + test + ' train data: %.2f, test data: %.2f' % (
    r2_score(train_y, y_train_pred),
    r2_score(test_y, y_test_pred)))
```

#Using function

```
rsquare_meansquare_error(y_train, y_test, X_train, X_test, "Regression tree", dtr_best)
```

#Visualize your tree USING the "best" parameteres/estimators

 # REQUIREMENTS:

```
# pip install pydotplus
# brew install graphviz

# Use graphviz to make a chart of the regression tree decision points:
from sklearn.externals.six import StringIO
from IPython.display import Image
from sklearn.tree import export_graphviz
import pydotplus
import pydot

dot_data = StringIO()
''' dtr_best was defined before in section B'''

## Graph
export_graphviz(dtr_best, out_file=dot_data,
                filled=True, rounded=True,
                special_characters=True,
                feature_names=X.columns)

graph = pydotplus.graph_from_dot_data(dot_data.getvalue())
Image(graph.create_png())
```

- **COMMANDS FOR RANDOM FOREST REGRESSION TREE**

```
from sklearn.ensemble import RandomForestRegressor
forest = RandomForestRegressor( )

params = {'max_depth':[3,4,5],
          'max_features':[2,3,4],
          'max_leaf_nodes':[5,6,7],
          'min_samples_split':[3,4],
         'n_estimators': [100]
         }

from sklearn.grid_search import GridSearchCV
estimator_rfr = GridSearchCV(forest, params, n_jobs=-1,  cv=5,verbose=1)
```

Notes: **** min_samples_leaf : The minimum number of samples required to be at a leaf node.***
**** 'max_leaf_nodes': The max number of leaves in the tree.***

#Fit the Random forest tree model

```
estimator_rfr.fit(X_train, y_train)
```

#Print best estimator, best parameters, and best score (best fit to explain Y)
''' rfr_best = is the random forest regression tree regressor with best parameters/estimators'''

```
rfr_best = estimator_rfr.best_estimator_
print "best estimator", rfr_best
print "\n==========\n"
print "best parameters", estimator_rfr.best_params_
print "\n==========\n"
print "best score", estimator_rfr.best_score_
```

#Using the function defined in Simple Reg Tree to Print feature Importance
```
feature_importance(X, rfr_best)
```

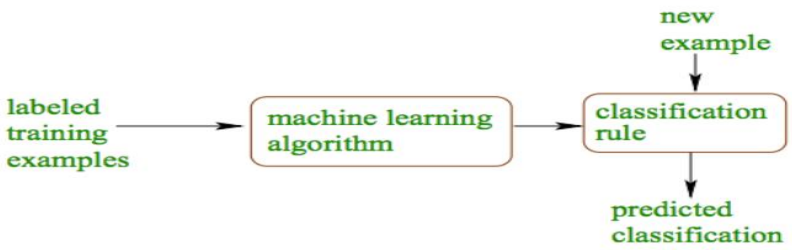
#Predict on the Test Data
```
y_pred_rfdtr= rfr_best.predict(X_test)
y_pred_rfdtr
```

#Evaluate the performance of your model (MSE in train and test data, R2 in train and test data) using function created above
```
rsquare_meansquare_error(y_train, y_test, X_train, X_test, "Random Forest Regression tree", rfr_best)
```

➢ CLASSIFICATION WITH LOGISTIC REGRESSION, CONFUSION MATRIX AND ROC CURVE

It is a regression, on which you estimate the probabilities of class membership. A binary outcome case with two classes, but can be generalized to multiple classes. Logistic regression is used in cases where the probability of an outcome (classification) is required.
Logistic regression application examples:
•text categorization (e.g., spam filtering)
•fraud detection
•machine vision (e.g., face detection)

Remember that Machine learning studies how to automatically learn to make accurate predictions based on past observations.



- **STEPS:**

•Define your target and predictors
•Split the data into train and test
•Implement a Gridsearch (you can add Lasso and Ridge)
•Fit the regression
•Find the best parameters and score (accuracy)
•What is your baseline?

➢ Command for Logistic Regression

Define the colum/variable/feature names

```
columns = [
   "class",
   "handicapped_infants",
   "water_project_cost",
   "adoption_of_the_budget_resolution"]
```

'''We are going to read the data directly from the web'''

```
csv_url = "http://archive.ics.uci.edu/ml/machine-learning-databases/voting-records/house-votes-84.data"
```

#Here we are reading the data and create a binary var 0 for republican 1 for democrat

```
house_df = pd.read_csv(csv_url, names = columns)
```

#Now we assign values of 0 if republican won, and 1 if lost

```
house_df['class'] = house_df['class'].map(lambda value: 0 if value == "republican" else 1 )
```

#Count number of values per class/category

```
house_df.immigration.value_counts()
```

#This creates the dataset into dummies

```
df_dummies = pd.get_dummies(house_df)
df_dummies.head(3)
```

#Define y and X

```
y = df_dummies['class']
columns_ = df_dummies.columns.tolist()
```

```
exclude_col = ['class']
X = df_dummies[[i for i in columns_ if i not in exclude_col]]
print (X.shape)
print (y.shape)
```

#Split the data

```
from sklearn.cross_validation import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.30, random_state=10)
print X_train.shape
print y_train.shape
print X_test.shape
print y_test.shape
```

#Call the logistic function

```
from sklearn.linear_model import LogisticRegression
logistic = LogisticRegression()
```

#set up parameters for the gridsearch I am using all the posible choices read documentation

```
Cs = np.logspace(0.0, 5.0, 20)
```

#WE ARE CREATING A DICTIONNARY THAT WILL FIND THE BEST MODEL FOR THE METHOD YOU CHOOSE (LASSO, RIDGE, ELASTICNET..)

```
search_parameters = {
        "penalty": ['l1','l2'],
        # Used to specify the norm used in the penalization.

        "C": Cs,  #Find the lambda value
        # Regularization parameter
        # "dual":[True, False], # Dual or primal formulation. Dual formulation is only implemented for
l2
        # penalty with liblinear solver. Prefer dual=False when n_samples > n_features

        "fit_intercept": [False, True],
        # Specifies if a constant (a.k.a. bias or intercept) should be added to the decision function.

        "class_weight": [None, "balanced"],
        # The “balanced” mode uses the values of y to automatically adjust weights inversely
        # proportional to class frequencies in the input data as n_samples / (n_classes *
np.bincount(y))

        "intercept_scaling": [2, 1],
        # Useful only if solver is liblinear. when self.fit_intercept is True, instance vector x becomes
        # [x, self.intercept_scaling], i.e. a “synthetic” feature with constant value equals to
intercept_scaling
```

```
            # is appended to the instance vector.

            "solver":  ['liblinear']
}
'''Gridsearch your parameters''' #This is used to cross validate your findings in different sets of data
from sklearn.grid_search import GridSearchCV
estimator = GridSearchCV(logistic, search_parameters, cv=5, verbose=1, n_jobs=-1)

#Fitting your data
estimator.fit(X_train, y_train)

##which one is the best estimator (ACCURACY SCORE) IN REGRESSION (LASSO, RIDGE...), WE CALCULATE MSE INSTEAD

best = estimator.best_estimator_
print "Best estimators on the left out data:\n", best
print "\nBest C / Regularization Param on the left out data:\n", estimator.best_estimator_.C

# This estimator.best_estimator_ object has many great reporting metrics
## Estimator that was chosen by the search, i.e.
## estimator which gave highest score (or smallest loss if specified) on the left out data.
## Not available if refit=False.

print "\nBest Params on hold out data (train):\n", estimator.best_params_
##Parameter setting that gave the best results on the hold out data.

print "\nBest Score on left out data:%.3f \n" % estimator.best_score_
## Score of best_estimator on the left out data.

#Baseline
#We know at baseline that 0 is republican 1 is democrat
print df_dummies['class'].value_counts(), "\n"
print "if I randomly choose, %.0f percent of the time I will be choosing democrat " % ((np.mean(df_dummies['class']))*100)
```

- **Logisitic Regression - Evaluation with Confusion Matrix**

•The terminology that is used to describe its components is a little hard to get it "at first."

- True Positive (tp) : The cases in which the model predicted "yes/positive", and the truth is also "yes/positive."
- True Negatives (tn) : The cases in which the model predicted "no/negative", and the truth is also "no/negative."
- False Positives (fp) : The cases in which the model predicted "yes/positive", and the truth is "no/negative".
- False Negatives (fn) : The cases in which the model predicted "no/negative", and the truth is "yes/positive".

		predicted	
		positive	negative
truth	positive	tp	fn
	negative	fp	tn

NB: IF WE HAVE A BINARY VARIABLE, TO EVALUATE, WE USE ROC METHOD, WITH MULTINOMIAL, WE USE CONFUSION MATRIX.

number of test examples *n = tp + tn + fp + fn*

Accuracy: In general how often is the classifier correct? => (*tp + tn*) / *n*

Misclassification Rate (Error Rate): How often is the model wrong => *fp + fn* / *n*

Precision: When the model predicts "yes", how often is it correct? => *tp* / (*tp + fp*)

Recall / True Positive Rate: How often the model predicts yes, when it's actually yes => *tp* / (*tp + fn*)

- **Logisitic Regression - Implement a Confusion Matrix**

 # Load Confusion Matrix package

```
from sklearn.metrics import confusion_matrix
```

#This is the simplistic way to run a confusion Matrix

```
y_pred=estimator.predict(X_test)
confmat = confusion_matrix(y_true=y_test, y_pred=y_pred)
confmat
```

#Pass it to a dataframe

```
confusion = pd.DataFrame(confmat, index=['True_Label_0 Republican', 'True_Label_1 Democrat'],
                         columns= ['Predict_Label_0 Republican', 'Predict_Label_1 Democrat'])
confusion
```

OR

#you can also define a function

```
def confus_mat(ytrue, ypred_method, class_0, class_1):
    confmat = confusion_matrix(y_true=ytrue, y_pred=ypred_method)

    confusion = pd.DataFrame(confmat, index=['True_Label_0 ' + class_0, 'True_Label_1 '+ class_1],
                             columns=['Predict_Label_0 ' + class_0, 'Predict_Label_1 ' + class_1])
    return confusion
```

#Call the function

```
y_pred=estimator.predict(X_test)
confus_mat(y_test, y_pred, "Republican", "Democrat" )
```

#function ix helps locating specific cells in the confusion matrix to use further for Error rate

```
TP = confusion.ix['True_Label_0 Republican', 'Predict_Label_0 Republican']
FP = confusion.ix['True_Label_1 Democrat', 'Predict_Label_0 Republican']
TN = confusion.ix['True_Label_1 Democrat', 'Predict_Label_1 Democrat']
FN = confusion.ix['True_Label_0 Republican', 'Predict_Label_1 Democrat']
print(zip(['True Positives','False Positives','True Negatives','False Negatives'],
      [TP, FP, TN, FN]))
```

- **Logisitic Regression - Calculate accuracy, Misclassification Rate (Error Rate), Precision, Recall**

Accuracy
How often is the classifier correct?

```
from sklearn.metrics import accuracy_score
acc = accuracy_score(y_test, y_pred)
print "Accuracy score: %.3f" %(acc*100)
```

Misclassification Rate (Error Rate)
How often is the model wrong

```
print "Error rate: %.3f" % (((FP + FN))/ float(len(y_test))*100)
```

Precision
Ability of the classifier to avoid labeling a class as a member of another class

```
from sklearn.metrics import precision_score
pcs = precision_score(y_test, y_pred)
print "Precision: %.3f" %(pcs*100)
```

 # precision score of 1 indicates that the classifier never mistakenly classified the current class as another class. precision score of 0 would mean that the classifier misclassified every instance of the current class

Recall
Recall the ability of the classifier to correctly identify the current class

```
from sklearn.metrics import recall_score
rcs = recall_score(y_test, y_pred)
print "Recall: %.3f" % (rcs*100)
```

 # A recall of 1 indicates that the classifier correctly predicted all observations. 0 means the classifier predicted all observations of the current class incorrectly.

#Print a classification report

```
from sklearn.metrics import classification_report
cls_rep = classification_report(y_test, y_pred)
print cls_rep
```

- **Logisitic Regression - Implement a ROC and find the AUC**

```
from sklearn.metrics import roc_curve, auc
```

 # Get out the predicted probabilities for the X_test matrix

```
y_pp = estimator.predict_proba(X_test)[:,1]

# roc_curve returns the false positive rate and true positive rates as the threshold changes
# takes in the y and the predicted probabilities of the positive class from your model.
fpr, tpr, _ = roc_curve(y_test, y_pp)
roc_auc = auc(fpr, tpr)

plt.figure(figsize=[9,9])
plt.plot(fpr, tpr, label='ROC curve (area = %0.2f)' % roc_auc, linewidth=10, color='g')
plt.plot([0, 1], [0, 1], linestyle='--', color='gray', linewidth=2)
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate', fontsize=16)
plt.ylabel('True Positive Rate', fontsize=16)
plt.title('Receiver operating characteristic curve', fontsize=20)
plt.legend(loc="lower right")
plt.show()
```

•Is the amout of space underneath the ROC curve, it shows how well your TPR and FPR is looking in the aggregate.
•The greater the area under the curve, the better higher the quality of the model
•The greater the area under the curve, the higher the ratio of true positives to false positives as the threshold becomes more lenient ▪ AUC = 0, BAD ▪ AUC = 1, GOOD
An AUC of 1.0 is a perfect model, where the classifier never makes a mistake.

#Conclusion
•A diagonal line shows that the classifier is making completely random guesses (50/50 chance).
•A perfect classifier is the one that shows a perfect trade-off between TPR and FPR (graphically TPR of 1 and FPR of 0).
•Worse than guessing = the blue line is below the dotted line.
•Mediocre classifier = lines that show depressions
•Good Classifier = the ideal scenario where there is a 'hump shaped' curve that is continually increasing.

LIST OF STEPS:

- Define Y and X
- Print shape (to check conformability)
- Train-Test split
- Import Model Lasso/Logistic/Tree
- Grid Search
- Fit (xtrain, ytrain)
- Find parameters
- Find best scores
- PREDICT
- EVALUATE the Model
- MSE/R-square

Or (depending on what we do):
Classification Method
Confusion Matrix
ROC