```
In [ ]: import pandas as pd
        import numpy as np
        import seaborn as sns
        import matplotlib.pyplot as plt

        %matplotlib inline
        %config InlineBackend.figure_format = 'retina'
```

read csv file or files

```
In [ ]: df = pd.read_csv('../../../data/eco/lightning/backup_06_14_2019.gz.out.c
        sv.gz',compression='gzip',
                          usecols = ['SN', 'LAST_SEEN'], parse_dates=['LAST_SEEN'
        ], dtype={'SN': str}, nrows = 100)

        lists = []
        path = '../../../data/eco/lightning'
        files = os.listdir(path)
        for afile in files:
            if not os.path.isdir(afile):
                print(afile)
                try:
                    df0 = pd.read_csv(path + '/' + afile, compression='gzip', us
        ecols = ['SN', 'LAST_SEEN'], dtype={'SN': str})
                except:
                    print('error')
                    continue
                lists.append(df0)
        df0 = pd.concat(lists)
```

check basic information

```
In [ ]: df.info()
        df.describe()
        df.dtypes
        df.columns

        df.isna().sum() # the number of NaNs for each column

        df.drop_duplicates(inplace = True)

        df[df.duplicated(subset=cols)]  # the rows with duplicated 'cols'

        df.loc['a', 'data']           # the cell with index 'a' at column 'data'

        dff.loc[dff['BAN'].isin(contact['BAN']), 'contact'] = 1
```

Handling Missing data

```
In [ ]: missing = df[df.isnull().any(1)]  # return the data frame of all rows co
ntaining NaN(s)
df[df['Col2'].isnull()]           # select rows with NaN in particular c
olumn?
df.dropna(inplace = True)         #  by default, inplace = False, return
a new dataframe, and df not modified.
df.fillna('unknown', inplace = True) # fill NaN with 'unknown'
df.fillna(0, inplace = True)         # fill NaN with 0
df.fillna({'col1' : 0, 'col2' : -1}, inplace = True)  # fill NaN differe
ntly for each column
df[col1].fillna(method='ffill', inplace = True)       # fill NaN with la
st valid value, forward fill
df[col2].fillna(method='bfill', inplace = True)       # backward fill
```

Convert column type and apply function on column

```
In [ ]: # convert Year '1985.0' to '1985'
df['Year'] = df['Year'].astype('str')
df['Year'] = df['Year'].apply(lambda x:x[:4] if x != 'unknown' else x) #
operate on each cell of a column
```

column manipulation

```
In [ ]: df['Year'].unique()
df['col'].nunique()

df[df.column.isin(list)]       # all rows in df with a match in list
df[(df.column.isin(list)) & (df[col2] > 7)]     # select rows.
df.index
list(df.index)  # convert index to list
df['cumsum'] = df['sale'].cumsum()   # make a new column with cumulative
sums of sales, usually first sort
df[col].apply(lambda x:x*2)     # apply function to column, make it mult
iply by 2
df.sort_values('mpg', ascending = True/False, inplace = True/False)
# order rows by values of a column
df.rename(columns = {'y': 'year'}, inplace = True/False)        # order
 rows by values of a column
df.reset_index(drop = True/False, inplace = True/False)         # reset
 index to DataFrame to row numbers, moving index to a new column. drop
here means whether to drop the reset index
df.drop(columns = ['Length', 'Height'], inplace = True/False)         # d
rop columns from DataFrame
```

datetime

In [ ]:
```python
df0['time'] = pd.to_datetime(df0['LAST_SEEN'])

df1['date'] = df1['time'].dt.date
```

correlation map

In [ ]:
```python
corrMatt = rides[['SPD', 'CLG', 'TEMP', 'DEWP', 'SLP','cnt']].corr()
mask = np.array(corrMatt)
mask[np.tril_indices_from(mask)] = False

fig,ax= plt.subplots()
fig.set_size_inches(20,10)

sns.heatmap(corrMatt, mask=mask,vmax=.8, square=True,annot=True, annot_kws={"size": 20}, linewidths=1)
ax.set_title('Correlation Heatmap', fontsize = 30)

fig.show()
```

reshape df

In [ ]:
```python
pd.melt(df, id_vars = ['Date'], value_vars=['Type', 'Value'],
        var_name = 'Variable', value_name = 'Observations')
```

group operation

In [ ]:
```python
df0 = df.groupby('order')["ext price"].sum()  # returns is not a data frame, the index now is 'order'
df0.reset_index(inplace=True)  # now it is dataframe, reset index to DataFrame to row numbers, moving index to column 'order'
df0 = df.groupby('order')[["ext price"]].sum()  # returns a dataframe, with index 'order'

df["Order_Total"] = df.groupby('order')["ext price"].transform('sum')

aggfunc ={'a': {'percentage':lambda x: len(x[x>0])/len(x)}}
temp.groupby('c').agg(aggfunc).reset_index()
```

plots

In [ ]:

```
# Distribution Plot
sns.distplot(df.Price, ax=ax2)

# Count Plot
ax = sns.countplot(x='Year', data = df, order = sorted(df['Year'].unique
()))
# or
sns.countplot(x='Year', data = df, order = sorted(df['Year'].unique()),
ax = ax1)
ax1 = sns.countplot(x='Genre', hue = 'Region', data = df, order = list(d
f['Genre'].value_counts().index))

# Box Plot
ax = sns.boxplot(x='Genre', y="Sales", hue = 'Region', data=df0, palette
=palette)
# or
ax = sns.boxplot(y="Sales")

# Bar Plot
ax = sns.barplot(x='Genre', y="Sales", hue = 'Region', data=df0, palette
=palette)

# Point Plot
ax = sns.pointplot(x='Year', y="Sales", hue = 'Region', data=df0, palett
e=palette)
```

In [ ]:

```
fig = plt.figure(figsize=(7,4))
ax = sns.distplot(np.log(rides.CLG+1))
ax.set_title("Log Transformed Distribution of 'CLG'", fontsize = 16)
ax.set_xlabel("Value", fontsize = 12)
ax.set_ylabel("Number of Records", fontsize = 12)
fig.show()
```

```
In [ ]: import matplotlib.pyplot as plt
        import seaborn as sns

         # 创建1幅图 ax
        fig, ax = plt.subplots(figsize=(16,18))

        # 创建2个子图x1 and ax2
        fig = plt.figure(figsize=(16,18))                   # change the fig
        size can make the title not overlap with the text

        ax1 = fig.add_subplot(121)
        ax2 = fig.add_subplot(122)

        # apply seaborn or plt to plot data on ax1  and ax2, we can plot bar cha
        rt, boxplot, point plot…
        # details and examples are in following sections.

        ax1.set_title('subplot title', fontsize=16, weight='bold')
        ax1.set_xlabel('xlabel', fontsize = 14)
        ax1.set_ylabel('ylabel', fontsize = 14)

        plt.suptitle('Title for all subplots', fontsize=18, weight='bold')  # ad
        d the title for all subplots if there are more than 2 subplots.

        plt.show()
```

Modelling

no time order

split data to features and target

```
In [ ]: y = df['NoShow']
        features_raw = df.drop(['NoShow','PatientID','AppointmentID','ScheduledD
        ay', 'AppointmentDay'], axis = 1)
```

log transform data

In [ ]:
```
fig = plt.figure(figsize=(16,5))

ax1 = fig.add_subplot(131)
ax2 = fig.add_subplot(132)
ax3 = fig.add_subplot(133)

sns.set_style("whitegrid")
sns.boxplot(y=df['units'], ax=ax1)
sns.distplot(df['units'], ax=ax2)
sns.distplot(np.log(df['units']), ax=ax3)

ax2.set(title="Skewed Distribution of Target Variable 'units'")
ax3.set(title="Log Transformed Distribution of Target Variable 'units'")

plt.show()
```

In [ ]:
```
skewed = ['Age','WaitDay']
features_raw[skewed] = df[skewed].apply(lambda x: np.log(x + 1))
```

normalize the data

In [ ]:
```
# Import sklearn.preprocessing.StandardScaler
from sklearn.preprocessing import MinMaxScaler

# Initialize a scaler, then apply it to the features
scaler = MinMaxScaler()
numerical = ['Age','WaitDay']
features_raw[numerical] = scaler.fit_transform(df[numerical])

# Show an example of a record with scaling applied
display(features_raw.head(n = 1))
```

In [ ]:
```
quant_features = ['casual', 'registered', 'cnt', 'SPD', 'CLG', 'TEMP',
'DEWP', 'SLP']
# Store scalings in a dictionary so we can convert back later
scaled_features = {}
for each in quant_features:
    mean, std = data[each].mean(), data[each].std()
    scaled_features[each] = [mean, std]
    data.loc[:, each] = (data[each] - mean)/std
```

randomly split the data into training, validation, testing

```
In [ ]: from sklearn.cross_validation import train_test_split

        # Split the 'features' and 'NoShow' data into training and testing sets
        X_train, X_test, y_train, y_test = train_test_split(features, y, test_si
        ze = 0.2, random_state = 0)

        # Show the results of the split
        print ("Training set has {} samples.".format(X_train.shape[0]))
        print ("Testing set has {} samples.".format(X_test.shape[0]))
```

Model Building - regression metric

```
In [ ]: from sklearn.metrics import mean_squared_error

        rfModel = RandomForestRegressor(n_estimators=100)
        rfModel.fit(X_train, y_train)
        preds_test = rfModel.predict(X_test)
        score = mean_squared_error(y_test, preds_test)

        print ("MSE Value For Random Forest: ",score)


        from sklearn import linear_model
        from sklearn.linear_model import Ridge
        from sklearn.metrics import mean_squared_error, r2_score

        regr = Ridge(alpha=1.0)

        regr.fit(X_train, y_train)

        # Make predictions using the testing set
        y_pred = regr.predict(X_test)

        # The coefficients
        print('Coefficients: \n', regr.coef_)
        # The mean squared error
        print("Mean squared error: %.2f"
              % mean_squared_error(y_test, y_pred))
        # Explained variance score: 1 is perfect prediction
        print('Variance score: %.2f' % r2_score(y_test, y_pred))
```

Model Building - classification metric

In [ ]:
```
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import fbeta_score, accuracy_score

rf_clf = RandomForestClassifier(random_state=0)
rf_clf.fit(X_train, y_train)

predictions_test = rf_clf.predict(X_test)
# print predictions_test
score = fbeta_score(y_test, predictions_test, beta = 1)
print score
```

In [ ]:
```
from sklearn.metrics import confusion_matrix, roc_auc_score, precision_s
core, recall_score, classification_report

print('the confusion matrix is')
print(confusion_matrix(y_test, predictions_test))

print('the auc-roc score is')
print(roc_auc_score(y_test, predictions_test))

print('the precision_score score is')
print(precision_score(y_test, predictions_test))

print('the recall_score score is')
print(recall_score(y_test, predictions_test))

print('the classification_report is')
print(classification_report(y_test, predictions_test))
```

tune parameters

In [ ]:
```
from sklearn.metrics import make_scorer
from sklearn.model_selection import GridSearchCV, KFold
from time import time

n_params = { 'n_estimators':[3,5,10,50],
              'criterion':['gini','entropy'],
              'max_depth': [3,4,5],
               'min_samples_split':[2,3,4,5],
            'min_samples_leaf':[1,2],
               'class_weight':['balanced',None]}


scorer = make_scorer(fbeta_score, beta=1)

# Perform grid search on the classifier using 'scorer' as the scoring me
thod
gsrf = GridSearchCV(rf_clf, n_params, cv= KFold(n_splits=5,shuffle=True
), scoring=scorer)

print "start"
start = time() # Get start time

# Fit the grid search object to the training data and find the optimal p
arameters
grid_fit = gsrf.fit(X_train, y_train)
end = time() # Get end time
print "finish"
t_elaps = end - start
print t_elaps

# Get the estimator
best_clf = gsrf.best_estimator_

# Make predictions using the optimized model
best_predictions = best_clf.predict(X_test)
print "\n"
print "Final F-score on the testing data: {:.4f}".format(fbeta_score(y_t
est, best_predictions, beta = 1))
print "\n"
print "The optimized model is"
print best_clf
```

In [ ]:
```
mean_train_score = np.array(-gsrf.cv_results_['mean_train_score'])
mean_train_score = mean_train_score.reshape(len(MAX_DEPTH_OPTIONS), len(
n_estimators)).T
mean_val_score = np.array(-gsrf.cv_results_['mean_test_score'])
mean_val_score = mean_val_score.reshape(len(MAX_DEPTH_OPTIONS), len(n_es
timators)).T

# MAX_DEPTH_OPTIONS = [7,9,11,13,15,17,19,21]
# n_estimators = [5,10,50,100]

fig = plt.figure(figsize=(15,10))

for i, n in enumerate(n_estimators):
    ax = fig.add_subplot(2, 2, i+1)
    ax.plot(np.array(MAX_DEPTH_OPTIONS), mean_train_score[i], 'o-', colo
r="r",
             label="Training error")
    ax.plot(np.array(MAX_DEPTH_OPTIONS), mean_val_score[i], 'o-', color=
"g",
             label="Validation error")
    ax.xaxis.set(ticks=MAX_DEPTH_OPTIONS)
    ax.set_title("n_estimators = '%d'"%(n), fontsize = 14)
    ax.legend(loc="best")
    ax.set_xlabel('Max Depth')
    ax.set_ylabel('Mean Squared Error')

fig.show()
```

Feature importance for random Forest

In [ ]:
```
feature_importances = pd.DataFrame(rfModel.feature_importances_,
                                   index = train_features.columns,
                                    columns=['importance']).sort_values(
'importance',ascending=False)
feature_importances.index.name='feature'
feature_importances.reset_index(inplace = True)
feature_importances.head(10)
```

time order

In [ ]:

```
test_data = data[data['date'] == 28]
test_data = data[-31*24:]

# Hold out the last 5 days or so of the remaining data as a validation s
et
val_data = data[data['date'].isin([23,24,25,26,27])]

# Remove the test and validation data from the orginial data set
train_data = data[data['date'] < 23]

# Separate the data into features and targets
target_fields = ['units']
train_features, train_targets = train_data.drop(target_fields, axis=1),
train_data[target_fields]
val_features, val_targets = val_data.drop(target_fields, axis=1), val_da
ta[target_fields]
test_features, test_targets = test_data.drop(target_fields, axis=1), tes
t_data[target_fields]

X_train = np.array(train_features)
y_train = np.array(train_targets['cnt'])
X_val = np.array(val_features)
y_val = np.array(val_targets['cnt'])
X_test = np.array(test_features)
y_test = np.array(test_targets['cnt'])

print(X_train.shape)
print(y_train.shape)
```

tune max depth

In [ ]:

```
from sklearn.model_selection import GridSearchCV, PredefinedSplit
from time import time

MAX_DEPTH_OPTIONS = [11,13,15,17,19,21,23,25,27,29]
num_estimators = [5,10,50,100]

n_params = {'n_estimators':num_estimators,
            'max_depth': MAX_DEPTH_OPTIONS}

my_validation_fold = []

for i in range(len(X_train)):
    my_validation_fold.append(-1)

for i in range(len(X_val)):
    my_validation_fold.append(0)

# Perform grid search on the classifier using 'scorer' as the scoring me
thod
gsrf = GridSearchCV(rfModel, n_params, cv= PredefinedSplit(test_fold=my_
validation_fold),
                    scoring='neg_mean_squared_error')
# gsrf = GridSearchCV(rfModel, n_params, cv= 3,
#                     scoring='neg_mean_squared_error')

print("start")
start = time() # Get start time

# Fit the grid search object to the training data and find the optimal p
arameters
grid_fit = gsrf.fit(np.append(X_train,X_val,axis=0), np.append(y_train,y
_val,axis=0))
end = time() # Get end time
print("finish")
t_elaps = end - start
print(t_elaps)

# Get the estimator
best_clf = gsrf.best_estimator_

# Make predictions using the optimized model
best_predictions = best_clf.predict(X_test)
print("\n")
print("Final score on the testing data: {:.4f}".format(MSE(y_test, best_
predictions)))
print("\n")
print("The optimized model is")
print(best_clf)
print("\n")
print("The best score on the validation set is:")
print(gsrf.best_score_)
```