Udacity - Intro to Machine Learning

Identify Fraud from Enron Email

- #### Rommel DeGuzman

Enron Scandal Summary

In early December 2001, innovative energy company Enron Corporation, a darling of Wall Street investors with $63.4 billion in assets, went bust. It was the largest bankruptcy in U.S. history. Some of the corporation's executives, including the CEO and chief financial officer, went to prison for fraud and other offenses. Shareholders hit the company with a 40 billion dollar lawsuit, and the company's auditor, Arthur Andersen, ceased doing business after losing many of its clients.

It was also a black mark on the U.S. stock market. At the time, most investors didn't see the prospect of massive financial fraud as a real risk when buying U.S.-listed stocks. "U.S. markets had long been the gold standard in transparency and compliance," says Jack Ablin, founding partner at Cresset Capital and a veteran of financial markets.

The company's collapse sent ripples through the financial system, with the government introducing a set of stringent regulations for auditors, accountants and senior executives, huge requirements for record keeping, and criminal penalties for securities laws violations. In turn, that has led in part to less choice for U.S. stock investors, and lower participation in stock ownership by individuals.

1. Goal

- This project aims to look into the Enron dataset using a machine learning algorithm to identify the POI (Persons of Interest) and non-POI employees based on the public Enron financial and email corpus. Enron was an energy company and the darling of Wall Street investors for years until it went bust due to fraud and other offenses.

Understanding the Dataset and Question

2. Dataset Exploration

> - **Data Exploration (related lesson: "Datasets and Questions")** - Student response addresses the most important characteristics of the dataset and uses these characteristics to inform their analysis. Important characteristics include:
>
> ```
>   - total number of data points<br>
>   - allocation across classes (POI/non-POI)<br>
>   - number of features used<br>
>   - are there features with many missing values? etc.<br>
> ```
>
> - **Outlier Investigation (related lesson: "Outliers")** - Student response identifies outlier(s) in the financial data, and explains how they are removed or otherwise handled

In [2]:

```
### Task 1: Select what features you'll use.
### features_list is a list of strings, each of which is a feature name.
### The first feature must be "poi".
```

```python
import sys
import pickle
sys.path.append("../tools/")
import numpy as np


from feature_format import featureFormat, targetFeatureSplit
from tester import dump_classifier_and_data

### Task 1: Select what features you'll use. Features_list is a list of strings, each of w
### The first feature must be "poi".

features_list = ['poi','salary','bonus', 'email_address', 'total_stock_value', 'expenses',
                 'restricted_stock','total_stock_value', 'exercised_stock_options','total_

financial_features = ['salary', 'total_payments', 'bonus', 'restricted_stock_deferred', 'c
'expenses', 'exercised_stock_options', 'other', 'long_term_incentive', 'restricted_stock',

email_features = ['to_messages', 'email_address', 'from_poi_to_this_person', 'from_message
                  'shared_receipt_with_poi']

POI_label = ['poi']

total_features = features_list

### Load the dictionary containing the dataset
with open("final_project_dataset.pkl", "rb") as data_file:
    enron_data = pickle.load(data_file)
```

In [4]:

```python
poi = 0
for name in enron_data.values():
    if name['poi']:
        poi += 1
print("number of poi: ", poi)
print("number of person who is not poi: ", len(enron_data) - poi)
```

```
number of poi:  18
number of person who is not poi:  128
```

In [5]:

```python
# Convert dataset to panda dataframe for each, then transpose
import pandas as pd
import numpy as np

df_enron = pd.DataFrame(enron_data)
df_enron = df_enron.transpose()
```

In [6]:

```python
# panda dataframe shape

df_enron.shape
```

Out[6]:

```
(146, 21)
```

- #### Dataset Information

> The dataset has **146** datapoints and **21** features, with **128** non-POIs and **18** POIs. In addition, it contained real email messages between senior management (poi's and non-poi's). Therefore, we can explore this dataset, identify email patterns, and investigate any correlations between salary bonuses within senior management.

In [7]:

```
#panda dataframe head

df_enron.head()
```

Out[7]:

	salary	to_messages	deferral_payments	total_payments	loan_advances	bonus	email_addres
METTS MARK	365788	807	NaN	1061827	NaN	600000	mark.metts@enron.com
BAXTER JOHN C	267102	NaN	1295738	5634343	NaN	1200000	NaN
ELLIOTT STEVEN	170941	NaN	NaN	211725	NaN	350000	steven.elliott@enron.com
CORDES WILLIAM R	NaN	764	NaN	NaN	NaN	NaN	bill.cordes@enron.com
HANNON KEVIN P	243293	1045	NaN	288682	NaN	1500000	kevin.hannon@enron.com

5 rows × 21 columns

In [8]:

```
# dataset data type info, prior to data type conversion

df_enron.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 146 entries, METTS MARK to GLISAN JR BEN F
Data columns (total 21 columns):
 #   Column                     Non-Null Count  Dtype 
---  ------                     --------------  ----- 
 0   salary                     146 non-null    object
 1   to_messages                146 non-null    object
 2   deferral_payments          146 non-null    object
 3   total_payments             146 non-null    object
 4   loan_advances              146 non-null    object
 5   bonus                      146 non-null    object
 6   email_address              146 non-null    object
 7   restricted_stock_deferred  146 non-null    object
 8   deferred_income            146 non-null    object
 9   total_stock_value          146 non-null    object
 10  expenses                   146 non-null    object
 11  from_poi_to_this_person    146 non-null    object
 12  exercised_stock_options    146 non-null    object
 13  from_messages              146 non-null    object
 14  other                      146 non-null    object
 15  from_this_person_to_poi    146 non-null    object
 16  poi                        146 non-null    object
 17  long_term_incentive        146 non-null    object
 18  shared_receipt_with_poi    146 non-null    object
 19  restricted_stock           146 non-null    object
 20  director_fees              146 non-null    object
dtypes: object(21)
memory usage: 25.1+ KB
```

In [9]:

```
# panda dataframe features value description

df_enron.describe()
```

Out[9]:

	salary	to_messages	deferral_payments	total_payments	loan_advances	bonus	email_address	restricted_sto
count	146	146	146	146	146	146	146	
unique	95	87	40	126	5	42	112	
top	NaN	NaN	NaN	NaN	NaN	NaN	NaN	
freq	51	60	107	21	142	64	35	

4 rows × 21 columns

- **I'll delete pandas dataframe features that I deemed unimportant from this exploration ('deferral_payments', loan_advances','restricted_stock_deferred','deferred_income','other','director_fees') and have more than 50% missing values ('NaN)**

In [10]:
```
df_enron.drop(['deferral_payments', 'loan_advances','restricted_stock_deferred','deferred_
```

- **Then convert pandas dataframe features data type for 'salary', 'total_payments', 'bonus', 'total_stock_value' and 'exercised_stock_options' to 'Float64'.**

In [11]:
```
df_enron["salary"] = df_enron.salary.astype(float)
df_enron["total_payments"] = df_enron.total_payments.astype(float)
df_enron["bonus"] = df_enron.bonus.astype(float)
df_enron["total_stock_value"] = df_enron.total_stock_value.astype(float)
df_enron["exercised_stock_options"] = df_enron.exercised_stock_options.astype(float)
df_enron["long_term_incentive"] = df_enron.long_term_incentive.astype(float)
```

- **Pandas dataframe after converting some columns data type to float64.**

In [12]:
```
df_enron.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 146 entries, METTS MARK to GLISAN JR BEN F
Data columns (total 15 columns):
 #   Column                   Non-Null Count  Dtype
---  ------                   --------------  -----
 0   salary                   95 non-null     float64
 1   to_messages              146 non-null    object
 2   total_payments           125 non-null    float64
 3   bonus                    82 non-null     float64
 4   email_address            146 non-null    object
 5   total_stock_value        126 non-null    float64
 6   expenses                 146 non-null    object
 7   from_poi_to_this_person  146 non-null    object
 8   exercised_stock_options  102 non-null    float64
 9   from_messages            146 non-null    object
 10  from_this_person_to_poi  146 non-null    object
 11  poi                      146 non-null    object
 12  long_term_incentive      66 non-null     float64
 13  shared_receipt_with_poi  146 non-null    object
 14  restricted_stock         146 non-null    object
dtypes: float64(6), object(9)
memory usage: 18.2+ KB
```

In [13]:
```
print('Number of datapoints before outliers removal: ', len(enron_data))
```

```
Number of datapoints before outliers removal:  146
```

- **I will remove any values that stand out from the sorted by names list below.**

In [14]:

```python
import pprint

pretty = pprint.PrettyPrinter()

names = sorted(enron_data.keys())

print('Enron employees sorted by last names')
pretty.pprint(names)
```

```
Enron employees sorted by last names
['ALLEN PHILLIP K',
 'BADUM JAMES P',
 'BANNANTINE JAMES M',
 'BAXTER JOHN C',
 'BAY FRANKLIN R',
 'BAZELIDES PHILIP J',
 'BECK SALLY W',
 'BELDEN TIMOTHY N',
 'BELFER ROBERT',
 'BERBERIAN DAVID',
 'BERGSIEKER RICHARD P',
 'BHATNAGAR SANJAY',
 'BIBI PHILIPPE A',
 'BLACHMAN JEREMY M',
 'BLAKE JR. NORMAN P',
 'BOWEN JR RAYMOND M',
 'BROWN MICHAEL',
 'BUCHANAN HAROLD G',
 'BUTTS ROBERT H',
 'BUY RICHARD B',
 'CALGER CHRISTOPHER F',
 'CARTER REBECCA C',
 'CAUSEY RICHARD A',
 'CHAN RONNIE',
 'CHRISTODOULOU DIOMEDES',
 'CLINE KENNETH W',
 'COLWELL WESLEY',
 'CORDES WILLIAM R',
 'COX DAVID',
 'CUMBERLAND MICHAEL S',
 'DEFFNER JOSEPH M',
 'DELAINEY DAVID W',
 'DERRICK JR. JAMES V',
 'DETMERING TIMOTHY J',
 'DIETRICH JANET R',
 'DIMICHELE RICHARD G',
 'DODSON KEITH',
 'DONAHUE JR JEFFREY M',
 'DUNCAN JOHN H',
 'DURAN WILLIAM D',
 'ECHOLS JOHN B',
 'ELLIOTT STEVEN',
 'FALLON JAMES B',
 'FASTOW ANDREW S',
 'FITZGERALD JAY L',
 'FOWLER PEGGY',
 'FOY JOE',
 'FREVERT MARK A',
 'FUGH JOHN L',
 'GAHN ROBERT S',
```

```
'GARLAND C KEVIN',
'GATHMANN WILLIAM D',
'GIBBS DANA R',
'GILLIS JOHN',
'GLISAN JR BEN F',
'GOLD JOSEPH',
'GRAMM WENDY L',
'GRAY RODNEY',
'HAEDICKE MARK E',
'HANNON KEVIN P',
'HAUG DAVID L',
'HAYES ROBERT E',
'HAYSLETT RODERICK J',
'HERMANN ROBERT J',
'HICKERSON GARY J',
'HIRKO JOSEPH',
'HORTON STANLEY C',
'HUGHES JAMES A',
'HUMPHREY GENE E',
'IZZO LAWRENCE L',
'JACKSON CHARLENE R',
'JAEDICKE ROBERT',
'KAMINSKI WINCENTY J',
'KEAN STEVEN J',
'KISHKILL JOSEPH G',
'KITCHEN LOUISE',
'KOENIG MARK E',
'KOPPER MICHAEL J',
'LAVORATO JOHN J',
'LAY KENNETH L',
'LEFF DANIEL P',
'LEMAISTRE CHARLES',
'LEWIS RICHARD',
'LINDHOLM TOD A',
'LOCKHART EUGENE E',
'LOWRY CHARLES P',
'MARTIN AMANDA K',
'MCCARTY DANNY J',
'MCCLELLAN GEORGE',
'MCCONNELL MICHAEL S',
'MCDONALD REBECCA',
'MCMAHON JEFFREY',
'MENDELSOHN JOHN',
'METTS MARK',
'MEYER JEROME J',
'MEYER ROCKFORD G',
'MORAN MICHAEL P',
'MORDAUNT KRISTINA M',
'MULLER MARK S',
'MURRAY JULIA H',
'NOLES JAMES L',
'OLSON CINDY K',
'OVERDYKE JR JERE C',
'PAI LOU L',
'PEREIRA PAULO V. FERRAZ',
'PICKERING MARK R',
'PIPER GREGORY F',
'PIRO JIM',
'POWERS WILLIAM',
'PRENTICE JAMES',
'REDMOND BRIAN L',
'REYNOLDS LAWRENCE',
'RICE KENNETH D',
'RIEKER PAULA H',
'SAVAGE FRANK',
'SCRIMSHAW MATTHEW',
```

```
 'SHANKMAN JEFFREY A',
 'SHAPIRO RICHARD S',
 'SHARP VICTORIA T',
 'SHELBY REX',
 'SHERRICK JEFFREY B',
 'SHERRIFF JOHN R',
 'SKILLING JEFFREY K',
 'STABLER FRANK',
 'SULLIVAN-SHAKLOVITZ COLLEEN',
 'SUNDE MARTIN',
 'TAYLOR MITCHELL S',
 'THE TRAVEL AGENCY IN THE PARK',
 'THORN TERENCE H',
 'TILNEY ELIZABETH A',
 'TOTAL',
 'UMANOFF ADAM S',
 'URQUHART JOHN A',
 'WAKEHAM JOHN',
 'WALLS JR ROBERT H',
 'WALTERS GARETH W',
 'WASAFF GEORGE',
 'WESTFAHL RICHARD K',
 'WHALEY DAVID A',
 'WHALLEY LAWRENCE G',
 'WHITE JR THOMAS E',
 'WINOKUR JR. HERBERT S',
 'WODRASKA JOHN',
 'WROBEL BRUCE',
 'YEAGER F SCOTT',
 'YEAP SOON']
```

> After reviewing the names of employees from the sorted list of employees by last name above, I noticed two values that are not valid names. These are **'THE TRAVEL AGENCY IN THE PARK'** and **'TOTAL'** and will also remove from the dataset.

In [15]:

```
enron_data.pop('THE TRAVEL AGENCY IN THE PARK',0)
enron_data.pop('TOTAL',0)
```

Out[15]:

```
{'salary': 26704229,
 'to_messages': 'NaN',
 'deferral_payments': 32083396,
 'total_payments': 309886585,
 'loan_advances': 83925000,
 'bonus': 97343619,
 'email_address': 'NaN',
 'restricted_stock_deferred': -7576788,
 'deferred_income': -27992891,
 'total_stock_value': 434509511,
 'expenses': 5235198,
 'from_poi_to_this_person': 'NaN',
 'exercised_stock_options': 311764000,
 'from_messages': 'NaN',
 'other': 42667589,
 'from_this_person_to_poi': 'NaN',
 'poi': False,
 'long_term_incentive': 48521928,
 'shared_receipt_with_poi': 'NaN',
 'restricted_stock': 130322299,
 'director_fees': 1398517}
```

- I want to review the list of employees values for **'total payments'** and **'total stock'** and remove the ones with empty/Nan values from both these features.

In [16]:

```
outliers =[]
for key in enron_data.keys():
    if  (enron_data[key]['total_payments']=='NaN') & (enron_data[key]['total_stock_value']
        outliers.append(key)
print ("Enron employees outliers:",(outliers))
```

```
Enron employees outliers: ['CHAN RONNIE', 'POWERS WILLIAM', 'LOCKHART EUGENE E']
```

> After running a query on employees with null values on **"Total Payments"** and **"Total Stock Values"** features from the dataset, it returned three employees with null values; therefore, I will remove them from the dataset. **'CHAN RONNIE'**, **'POWERS WILLIAM'**, and **LOCKHART EUGENE E'** from the dataset.

In [17]:

```
enron_data.pop('CHAN RONNIE',0)
enron_data.pop('POWERS WILLIAM',0)
enron_data.pop('LOCKHART EUGENE E',0)
```

Out[17]:

```
{'salary': 'NaN',
 'to_messages': 'NaN',
 'deferral_payments': 'NaN',
 'total_payments': 'NaN',
 'loan_advances': 'NaN',
 'bonus': 'NaN',
 'email_address': 'NaN',
 'restricted_stock_deferred': 'NaN',
 'deferred_income': 'NaN',
 'total_stock_value': 'NaN',
 'expenses': 'NaN',
 'from_poi_to_this_person': 'NaN',
 'exercised_stock_options': 'NaN',
 'from_messages': 'NaN',
 'other': 'NaN',
 'from_this_person_to_poi': 'NaN',
 'poi': False,
 'long_term_incentive': 'NaN',
 'shared_receipt_with_poi': 'NaN',
 'restricted_stock': 'NaN',
 'director_fees': 'NaN'}
```

In [18]:

```
print('Number of people after outliers removal: ', len(enron_data))
```

```
Number of people after outliers removal:  141
```

- **I will then list the names of POI's from the dataset.**

In [19]:

```
df_enron[df_enron['poi'] == True]
```

Out[19]:

	salary	to_messages	total_payments	bonus	email_address	total_stock_value	ex
HANNON KEVIN P	243293.0	1045	288682.0	1500000.0	kevin.hannon@enron.com	6391065.0	
COLWELL WESLEY	288542.0	1758	1490344.0	1200000.0	wes.colwell@enron.com	698242.0	
RIEKER PAULA H	249201.0	1328	1099100.0	700000.0	paula.rieker@enron.com	1918887.0	

	salary	to_messages	total_payments	bonus	email_address	total_stock_value	e
KOPPER MICHAEL J	224305.0	NaN	2652612.0	800000.0	michael.kopper@enron.com	985032.0	
SHELBY REX	211844.0	225	2003885.0	200000.0	rex.shelby@enron.com	2493616.0	
DELAINEY DAVID W	365163.0	3093	4747979.0	3000000.0	david.delainey@enron.com	3614261.0	
LAY KENNETH L	1072321.0	4273	103559793.0	7000000.0	kenneth.lay@enron.com	49110078.0	
BOWEN JR RAYMOND M	278601.0	1858	2669589.0	1350000.0	raymond.bowen@enron.com	252055.0	
BELDEN TIMOTHY N	213999.0	7991	5501630.0	5249999.0	tim.belden@enron.com	1110705.0	
FASTOW ANDREW S	440698.0	NaN	2424083.0	1300000.0	andrew.fastow@enron.com	1794412.0	
CALGER CHRISTOPHER F	240189.0	2598	1639297.0	1250000.0	christopher.calger@enron.com	126027.0	
RICE KENNETH D	420636.0	905	505050.0	1750000.0	ken.rice@enron.com	22542539.0	
SKILLING JEFFREY K	1111258.0	3627	8682716.0	5600000.0	jeff.skilling@enron.com	26093672.0	
YEAGER F SCOTT	158403.0	NaN	360300.0	NaN	scott.yeager@enron.com	11884758.0	
HIRKO JOSEPH	NaN	NaN	91093.0	NaN	joe.hirko@enron.com	30766064.0	
KOENIG MARK E	309946.0	2374	1587421.0	700000.0	mark.koenig@enron.com	1920055.0	
CAUSEY RICHARD A	415189.0	1892	1868758.0	1000000.0	richard.causey@enron.com	2502063.0	
GLISAN JR BEN F	274975.0	873	1272284.0	600000.0	ben.glisan@enron.com	778546.0	

- **Then show the data statistics of these poi's.**

In [20]:

```
df_enron[df_enron['poi'] == True].describe()
```

Out[20]:

	salary	total_payments	bonus	total_stock_value	exercised_stock_options	long_term_incentive
count	1.700000e+01	1.800000e+01	1.600000e+01	1.800000e+01	1.200000e+01	1.200000e+01
mean	3.834449e+05	7.913590e+06	2.075000e+06	9.165671e+06	1.046379e+07	1.204862e+06
std	2.783597e+05	2.396549e+07	2.047437e+06	1.384117e+07	1.238259e+07	9.916583e+05
min	1.584030e+05	9.109300e+04	2.000000e+05	1.260270e+05	3.847280e+05	7.102300e+04
25%	2.401890e+05	1.142396e+06	7.750000e+05	1.016450e+06	1.456581e+06	3.689780e+05
50%	2.786010e+05	1.754028e+06	1.275000e+06	2.206836e+06	3.914557e+06	1.134637e+06
75%	4.151890e+05	2.665345e+06	2.062500e+06	1.051133e+07	1.938604e+07	1.646772e+06
max	1.111258e+06	1.035598e+08	7.000000e+06	4.911008e+07	3.434838e+07	3.600000e+06

- #### Let's look into how many poi's are there with missing values and how many are there

In [21]:
```
df_enron[df_enron["poi"] == True].isnull().sum()
```

Out[21]:
```
salary                       1
to_messages                  0
total_payments               0
bonus                        2
email_address                0
total_stock_value            0
expenses                     0
from_poi_to_this_person      0
exercised_stock_options      6
from_messages                0
from_this_person_to_poi      0
poi                          0
long_term_incentive          6
shared_receipt_with_poi      0
restricted_stock             0
dtype: int64
```

Optimize Feature Selection/Engineering

> - **Create new features (related lesson: "Feature Selection")** - At least one new feature is implemented. Justification for that feature is provided in the written response. The effect of that feature on final algorithm performance is tested or its strength is compared to other features in feature selection. The student is not required to include their new feature in their final feature set.
> - **Intelligently select features (related lesson: "Feature Selection")** - Univariate or recursive feature selection is deployed, or features are selected by hand (different combinations of features are attempted, and the performance is documented for each one). Features that are selected are reported and the number of features selected is justified. For an algorithm that supports getting the feature importances (e.g. decision tree) or feature scores (e.g. SelectKBest), those are documented as well.
> - **Properly scale features (related lesson: "Feature Scaling")** - If algorithm calls for scaled features, feature scaling is deployed.

- **I will be creating two new features to represent the message ratios of emails coming from poi's (fraction_from_poi) and message ratios of emails sent to poi's (fraction_to_poi). Then pass these new features to the SelectKBest function for feature selection.**

- ***These two new features will represent the ratios of the emails from poi (fraction_from_poi) to this person divided with all the other emails sent to person. And ratios of emails from this person to poi (fraction_to_poi) divided with all the emails from this person.***

- **Univariate feature selection works best through selecting the best features from a statistical tests. I utilized an automated feature selection function named SelectKBest. Tuning the parameter k (number of features) while also tuning the parameters of machine learning algorithm when implementing cross-validation. Selecting all 21 features from the dataset while tuning the parameters of the machine learning can result to overfitting.**

In [22]:
```
def computeFraction(poi_messages, all_messages):
```

```
    """ given a number messages to/from POI (numerator)
        and number of all messages to/from a person (denominator),
        return the fraction of messages to/from that person
        that are from/to a POI
   """
    fraction = 0.

    if poi_messages=='NaN' or all_messages=='NaN':
        fraction = 0.
    else:
        fraction=float(poi_messages)/float(all_messages)

    return fraction


for name in enron_data:

    data_point = enron_data[name]

    from_poi_to_this_person = data_point["from_poi_to_this_person"]
    to_messages = data_point["to_messages"]
    fraction_from_poi = computeFraction(from_poi_to_this_person, to_messages )
    data_point["fraction_from_poi"] = fraction_from_poi


    from_this_person_to_poi = data_point["from_this_person_to_poi"]
    from_messages = data_point["from_messages"]
    fraction_to_poi = computeFraction(from_this_person_to_poi, from_messages )
    data_point["fraction_to_poi"] = fraction_to_poi

features_list2 = total_features
features_list2.remove('email_address')
features_list2 =  features_list2 + ['fraction_from_poi', 'fraction_to_poi']
```

In [23]:

```
fraction_to_poi =[enron_data[key]["fraction_to_poi"] for key in enron_data]
fraction_from_poi=[enron_data[key]["fraction_from_poi"] for key in enron_data]
poi=[enron_data[key]["poi"]==1 for key in enron_data]
```

In [24]:

```
def Second(elem):
    """ sorted second element
    """
    return elem[1]
```

In [25]:

```
import matplotlib.pyplot as plt
from feature_format import featureFormat

def dict_to_list(key,normalizer):
    my_list=[]

    for i in enron_data:
        if enron_data[i][key]=="NaN" or enron_data[i][normalizer]=="NaN":
            my_list.append(0.)
        elif enron_data[i][key]>=0:
            my_list.append(float(enron_data[i][key])/float(enron_data[i][normalizer]))
    return my_list

### create two lists of new features
fraction_from_poi = dict_to_list("from_poi_to_this_person","to_messages")
fraction_to_poi = dict_to_list("from_this_person_to_poi","from_messages")
```

```
### insert new features into data_dict
count=0
for i in enron_data:
    enron_data[i]["fraction_from_poi"]=fraction_from_poi[count]
    enron_data[i]["fraction_to_poi"]=fraction_to_poi[count]
    count +=1


new_features = ["poi", "fraction_from_poi", "fraction_to_poi"]

    ### store to my_dataset for easy export below
my_dataset = enron_data
```

In [26]:

```
from feature_format import targetFeatureSplit

data = featureFormat(my_dataset, features_list2, sort_keys = True)
labels, features = targetFeatureSplit(data)
```

In [27]:

```
# intelligently select features (univariate feature selection)
from sklearn.feature_selection import SelectKBest, f_classif
import warnings
warnings.filterwarnings('ignore', category=RuntimeWarning)
selector = SelectKBest(f_classif, k = 13)
selector.fit(features, labels)
scores = zip(features_list2[1:], selector.scores_)
sorted_scores = sorted(scores, key = Second, reverse = True)
#pprint.pprint('SelectKBest scores: ')
pprint.pprint( sorted_scores)
all_features = POI_label + [(i[0]) for i in sorted_scores[0:20]]
#pprint.pprint( all_features)
SelectKBest_features = POI_label + [(i[0]) for i in sorted_scores[0:10]]
#pprint.pprint( 'KBest')
pprint.pprint( SelectKBest_features)
#print(my_dataset)
for emp in enron_data:
     for f in enron_data[emp]:
         if enron_data[emp][f] == 'NaN':
             # fill NaN values
             enron_data[emp][f] = 0
my_dataset = enron_data
```

```
[('exercised_stock_options', 24.25047235452619),
 ('total_stock_value', 23.613740454440887),
 ('bonus', 20.25718499812395),
 ('salary', 17.71787357924329),
 ('fraction_to_poi', 15.946248696687636),
 ('deferred_income', 11.222175285805182),
 ('long_term_incentive', 9.62221216430468),
 ('restricted_stock', 8.947938884292649),
 ('total_payments', 8.570823078730976),
 ('shared_receipt_with_poi', 8.277457991443601),
 ('expenses', 5.815328001904854),
 ('from_poi_to_this_person', 5.041257378669385),
 ('other', 4.070343006434408),
 ('fraction_from_poi', 2.963990314926164),
 ('from_this_person_to_poi', 2.295183195738003),
 ('to_messages', 1.5634425546665922),
 ('from_messages', 0.18071817710224855),
 ('restricted_stock_deferred', 0.06696644496108223)]
['poi',
 'exercised_stock_options',
 'total_stock_value',
 'bonus',
```

```
'salary',
'fraction_to_poi',
'deferred_income',
'long_term_incentive',
'restricted_stock',
'total_payments',
'shared_receipt_with_poi']
```

- ### Feature Scaling

Some of these features have different units and significant values and would be transformed by using sklearn **MinMaxScaler** to a given range of between **0** and **1**.

In [28]:

```
# dataset using original features
from sklearn import preprocessing
data = featureFormat(my_dataset, SelectKBest_features, sort_keys = True)
labels, features = targetFeatureSplit(data)
scaler = preprocessing.MinMaxScaler()
features = scaler.fit_transform(features)
```

- **I am utilizing an automated feature function SelectKBest from sklearn to select the best K features.**

In [29]:

```
# dataset with new added features
SelectKBest_features = SelectKBest_features + ['fraction_from_poi', 'fraction_to_poi']
data = featureFormat(my_dataset, SelectKBest_features, sort_keys = True)
new_labels, new_features = targetFeatureSplit(data)
new_features = scaler.fit_transform(new_features)
```

In [30]:

```
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, precision_score, recall_score

def tune_params(grid_search, features, labels, params, iters = 80):
    """ given a grid_search and parameters list (if exist) for a specific model,
    along with features and labels list,
    it tunes the algorithm using grid search and prints out the average evaluation metrics
    results (accuracy, percision, recall) after performing the tuning for iter times,
    and the best hyperparameters for the model
    """
    acc = []
    pre = []
    recall = []

    for i in range(iters):
        features_train, features_test, labels_train, labels_test = \
        train_test_split(features, labels, test_size = 0.3, random_state = i)
        grid_search.fit(features_train, labels_train)
        predicts = grid_search.predict(features_test)

        acc = acc + [accuracy_score(labels_test, predicts)]
        pre = pre + [precision_score(labels_test, predicts)]
        recall = recall + [recall_score(labels_test, predicts)]
    print ("accuracy: {}".format(np.mean(acc)))
    print ("precision: {}".format(np.mean(pre)))
    print ("recall: {}".format(np.mean(recall)))

    best_params = grid_search.best_estimator_.get_params()
    for param_name in params.keys():
        print("%s = %r, " % (param_name, best_params[param_name]))
```

1. Support Vector Machines

In [31]:
```
from sklearn.model_selection import GridSearchCV
import warnings
warnings.filterwarnings('ignore')
from sklearn import svm


svm_clf = svm.SVC()
svm_param = {'kernel':('linear', 'rbf', 'sigmoid'),
'gamma': [1, 0.1, 0.01, 0.001, 0.0001],
'C': [0.1, 1, 10, 100, 1000]}
svm_grid_search = GridSearchCV(estimator = svm_clf, param_grid = svm_param)


print("SVM model evaluation with Original Features")
tune_params(svm_grid_search, features, labels, svm_param)
print("SVM model evaluation with New Features")
tune_params(svm_grid_search, new_features, new_labels, svm_param)
```

```
SVM model evaluation with Original Features
accuracy: 0.8636627906976744
precision: 0.21065476190476193
recall: 0.08765873015873016
kernel = 'sigmoid',
gamma = 1,
C = 100,
SVM model evaluation with New Features
accuracy: 0.8642441860465115
precision: 0.15729166666666666
recall: 0.057668650793650786
kernel = 'sigmoid',
gamma = 1,
C = 10,
```

2. Decision Tree

In [32]:
```
from sklearn import tree
dt_clf = tree.DecisionTreeClassifier()
dt_param = {'criterion':('gini', 'entropy'),
'splitter':('best','random')}
dt_grid_search = GridSearchCV(estimator = dt_clf, param_grid = dt_param)

print("Decision Tree model evaluation with Original Features")
tune_params(dt_grid_search, features, labels, dt_param)
print("Decision Tree model evaluation with New Features")
tune_params(dt_grid_search, new_features, new_labels, dt_param)
```

```
Decision Tree model evaluation with Original Features
accuracy: 0.8261627906976743
precision: 0.33407151875901875
recall: 0.3296875
criterion = 'entropy',
splitter = 'random',
Decision Tree model evaluation with New Features
accuracy: 0.8180232558139535
precision: 0.3017135642135642
recall: 0.2996577380952381
criterion = 'gini',
splitter = 'random',
```

3. Naive Bayes

In [33]:

```python
import warnings
warnings.filterwarnings('ignore')
from sklearn.model_selection import GridSearchCV

from sklearn.naive_bayes import GaussianNB
nb_clf = GaussianNB()
nb_param = {}
nb_grid_search = GridSearchCV(estimator = nb_clf, param_grid = nb_param)
print("Naive Bayes model evaluation with Original Features")
tune_params(nb_grid_search, features, labels, nb_param)
print("Naive Bayes model evaluation with new features")
tune_params(nb_grid_search, new_features, new_labels, nb_param)
```

```
Naive Bayes model evaluation with Original Features
accuracy: 0.8470930232558139
precision: 0.3991815476190476
recall: 0.33036706349206346
Naive Bayes model evaluation with new features
accuracy: 0.8325581395348836
precision: 0.35005005411255413
recall: 0.31953373015873016
```

> **SVM model evaluation with Original Features**
>
> *Precision = 0.21065476190476193*
>
> *Recall = 0.08765873015873016*
>
> **SVM model evaluation with New Features**
>
> *Precision = 0.15729166666666666*
>
> *Recall = 0.057668650793650786*

> **Decision Tree model evaluation with Original Features**
>
> *Precision = 0.30395202020202017*
>
> *Recall = 0.2973759920634921*
>
> **Decision Tree model evaluation with New Features**
>
> *Precision = 0.3153879453879454*
>
> *Recall = 0.31696924603174603*

> **Naive Bayes model evaluation with Original Features**
>
> *Precision = 0.3991815476190476*
>
> *Recall = 0.33036706349206346*
>
> **Naive Bayes model evaluation with New Features**
>
> *Precision = 0.35005005411255413*
>
> *Recall = 0.31953373015873016*

Note: As you can see above, these algorithmns varies in performance from one algorithm to another. Some of them performed better when adding the new features such as Decision Trees. While SVM and Naive Bayes performed worse after adding these new features.

I then used the first 10 features(k=10) plus the POI to get the highest scores from SelectKBest

Pick and Tune an Algorithm

> - **Pick an algorithm (related lessons: "Naive Bayes" through "Choose Your Own Algorithm")** - At least two different algorithms are attempted and their performance is compared, with the best performing one used in the final analysis.

> - **Discuss parameter tuning and its importance** - Response addresses what it means to perform parameter tuning and why it is important.
> - **Tune the algorithm (related lesson: "Validation")** - At least one important parameter tuned with at least 3 settings investigated systematically, or any of the following are true:

- I tried three different algorithms and have decided to use **Naive Bayes** since it got the highest evaluation score. I also tried **"SVM"** and **"Decision Tree"****. These algorithms all showed higher accuracy scores and are probably not the best metric to use.

- Since most algorithms have multiple default values, tuning the classifier's specific parameters can help optimize its performance; otherwise, the data model can either be overfitting or underfitting. So, for example, we adjust SVM's hyperparameter '**kernel**', '**gamma**', and '**C**' to achieve the best possible performance. This is called hyperparameter optimization. It is an essential step in machine learning before the presentation.

- I have used sklearn's **GridSearchCV** library function as parameter tuning. They are used in **SVM** and **Decision Tree** algorithm. **GridSearchCV** implements a fit and score method and evaluate a model in each specified parameter combination.

Validate and Evaluate

> - **Usage of Evaluation Metrics (related lesson: "Evaluation Metrics")** - At least two appropriate metrics are used to evaluate algorithm performance (e.g. precision and recall), and the student articulates what those metrics measure in context of the project task.
> - **Discuss validation and its importance** - Response addresses what validation is and why it is important.
> - **Validation Strategy (related lesson "Validation")** - Performance of the final algorithm selected is assessed by splitting the data into training and testing sets or through the use of cross validation, noting the specific type of validation performed.
> - **Algorithm Performance** - When tester.py is used to evaluate performance, precision and recall are both at least 0.3.

- We use validation to evaluate the classifier using its training and testing dataset. We use it to measure its reliability and accuracy. If we train and test the classifier with the same data, it will yield overfitting results, so validation is essential. I will use StratifiedShuffleSplit to split the data between the training and testing datasets. This will guarantee that the classes are randomly selected and correctly allocated.

In [34]:

```
from sklearn import preprocessing
data = featureFormat(my_dataset, all_features, sort_keys = True)
labels1, new_features = targetFeatureSplit(data)
scaler = preprocessing.MinMaxScaler()
new_features = scaler.fit_transform(new_features)
```

In [35]:

```
from sklearn.feature_selection import SelectKBest, f_classif
def feature_selection(nb_features,features, labels):
    selector = SelectKBest(f_classif, k=nb_features)
    selector.fit(features, labels)
    return selector
```

In [36]:
```
from sklearn.model_selection import StratifiedShuffleSplit
from sklearn.feature_selection import SelectKBest, f_classif

def test_classifier(clf, labels, features, nb_features, folds = 1000):
    cv = StratifiedShuffleSplit(n_splits=folds, random_state=42)
    true_negatives = 0
    false_negatives = 0
    true_positives = 0
    false_positives = 0
    precision=0
    recall=0
    f1=0
    f2=0
    for train_idx, test_idx in cv.split(features, labels):
        features_train = []
        features_test  = []
        labels_train   = []
        labels_test    = []
        for ii in train_idx:
            features_train.append( features[ii] )
            labels_train.append( labels[ii] )
        for jj in test_idx:
            features_test.append( features[jj] )
            labels_test.append( labels[jj] )

        #Selection of the best K features
       # selector=feature_selection(nb_features,features_train, labels_train)
        selector=feature_selection(nb_features,features_train, labels_train)
        features_train_transformed = selector.transform(features_train)
        features_test_transformed  = selector.transform(features_test)

        ### fit the classifier using training set, and test on test set
        clf.fit(features_train_transformed, labels_train)
        predictions = clf.predict(features_test_transformed)
        for prediction, truth in zip(predictions, labels_test):
            if prediction == 0 and truth == 0:
                true_negatives += 1
            elif prediction == 0 and truth == 1:
                false_negatives += 1
            elif prediction == 1 and truth == 0:
                false_positives += 1
            elif prediction == 1 and truth == 1:
                true_positives += 1
            else:
                break

    try:
        total_predictions = true_negatives + false_negatives + false_positives + true_posi
        accuracy = 1.0*(true_positives + true_negatives)/total_predictions
        precision = 1.0*true_positives/(true_positives+false_positives)
        recall = 1.0*true_positives/(true_positives+false_negatives)
        f1 = 2.0 * true_positives/(2*true_positives + false_positives+false_negatives)
        f2 = (1+2.0*2.0) * precision*recall/(4*precision + recall)
    except:
        None
    return precision,recall,f1,f2
```

Evaluation metrics

I will use the two evaluation metrics **Precision and Recall**, which is used best in measuring prediction success with highly imbalanced classes. When retrieving information, **precision** measures the relevance of its result, while **recall** measures how accurately relevant are its results.

F1 scores measures the weighted average of precision and recall.

If the precision score is **0.51**, then it means there is a **51%** chance that the predicted POIs are truly POIs

If the recall score is **0.42**, then it means there is a **42%** chance that the POIs were identified correctly.

Choosing and tuning the algorithm

Decission Tree Classifier

> **Decision Trees (DTs) are a non-parametric supervised learning method used for classification and regression. The goal is to create a model that predicts the value of a target variable by learning simple decision rules inferred from the data features.**

In [37]:

```
#  Make a note of the different metrics

from sklearn import tree

nb_features_orig=len(new_features[1])

precision_result=[]
recall_result=[]
f1_result=[]
f2_result=[]
nb_feature_store=[]

dt_param = {'criterion':('gini', 'entropy'),
'splitter':('best','random')}

# calculate
for nb_features in range(1,nb_features_orig+1):
    #Number of neighbours

        #classifier
        clf = tree.DecisionTreeClassifier()
        #Cross-validate then calculate it's precision and recall metrics
        precision,recall,f1,f2=test_classifier(clf, labels1, new_features,nb_features, fo
        # Note each evaluation metrics
        precision_result.append(precision)
        recall_result.append(recall)
        f1_result.append(f1)
        f2_result.append(f2)
        nb_feature_store.append(nb_features)
```

In [38]:

```
import pandas as pd
result=pd.DataFrame([nb_feature_store,precision_result,recall_result,f1_result,f2_result])
result.columns=['nb_feature','precision','recall','f1','f2']
result.head()
```

Out[38]:

	nb_feature	precision	recall	f1	f2
0	1.0	0.213971	0.1455	0.173214	0.155449
1	2.0	0.185989	0.1885	0.187236	0.187992
2	3.0	0.282211	0.3165	0.298374	0.308992
3	4.0	0.272356	0.3000	0.285510	0.294031
4	5.0	0.257806	0.2890	0.272513	0.282171

Gaussian Naive Bayes (GaussianNB)

In [39]:

```
from sklearn.naive_bayes import GaussianNB
from sklearn.model_selection import StratifiedShuffleSplit
from sklearn.feature_selection import SelectKBest, f_classif
nb_features_orig=len(new_features[1])

precision_result=[]
recall_result=[]
f1_result=[]
f2_result=[]
nb_feature_store=[]

#Classifier
clf=GaussianNB()
#calculate the evaluation metrics for k best number of features selected in the model.
for nb_features in range(1,13):
    # Cross-validation and calculate precision and recall metrics
    precision,recall,f1,f2=test_classifier(clf, labels1, new_features, nb_features, folds
    # Note each evaluation metrics
    precision_result.append(precision)
    recall_result.append(recall)
    f1_result.append(f1)
    f2_result.append(f2)
    nb_feature_store.append(nb_features)
```

In [40]:

```
import pandas as pd
result=pd.DataFrame([nb_feature_store,precision_result,recall_result,f1_result,f2_result])
result.columns=['nb_feature','precision','recall','f1','f2']
result.head(10)
```

Out[40]:

	nb_feature	precision	recall	f1	f2
0	1.0	0.269231	0.1400	0.184211	0.154867
1	2.0	0.292969	0.1875	0.228659	0.202047
2	3.0	0.324343	0.2405	0.276199	0.253612
3	4.0	0.374517	0.2910	0.327518	0.304584
4	5.0	0.404439	0.3280	0.362231	0.340885
5	6.0	0.376881	0.3130	0.341983	0.323983
6	7.0	0.365940	0.3105	0.335948	0.320202
7	8.0	0.342806	0.2895	0.313906	0.298792
8	9.0	0.346109	0.2980	0.320258	0.306521
9	10.0	0.351351	0.3120	0.330508	0.319149

- **Summary:**

I was able to achieve the precision and recall of at least 0.3 using Naive Bayes and Decision Tree, although Naive Bayes ran a little faster the Decision Tree.

In [40]:

```
from tester import dump_classifier_and_data

### Task 6: Dump your classifier, dataset, and features_list
```

```
features_list = total_features
dump_classifier_and_data(clf, my_dataset, features_list)
```

References:

https://time.com/6125253/enron-scandal-changed-american-business-forever/

https://scikit-learn.org/stable/modules/tree.html#:~:text=Decision%20Trees%20(DTs)%20are%20a,as%20a%20piecewise%20const