Economic Health Analysis

Mark Styx | IT542 | Final Project | 3/18/2020

Code: https://github.com/meow1928/economic_health/blob/master/analysis.py (https://github.com/meow1928/economic_health/blob/master/analysis.py)

Collect the headers and summary data output from HDFS/Spark

In [3]:
```
import os
os.chdir('S:\Anchor\economic_health')
```

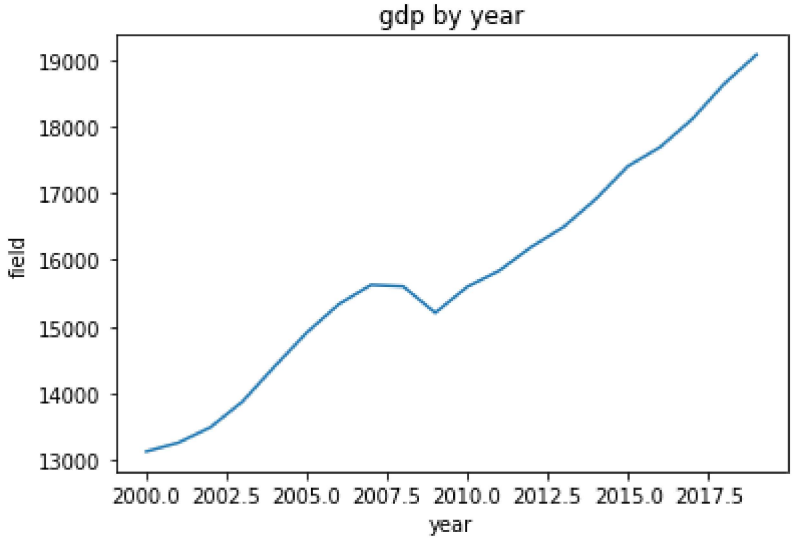
In [4]:
```
import pandas as pd

#get headers
with open('headers.txt','r') as f:
    headers = f.read()
headers = headers.split(',')
headers = [x for x in headers if x != '']

#load data
df = pd.read_csv('summary.csv',names=headers)

#normalize data types
for field in df.columns:
    df[field] = pd.to_numeric(df[field],errors='coerce')
```

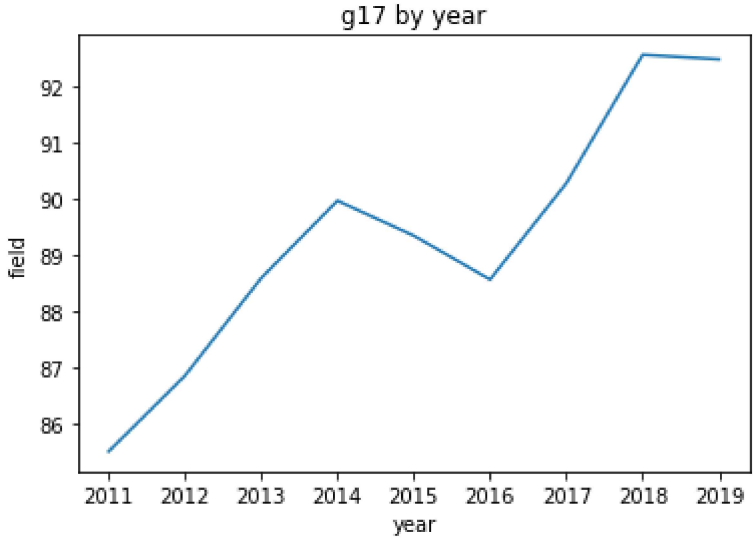
Create a function to generate graphs:

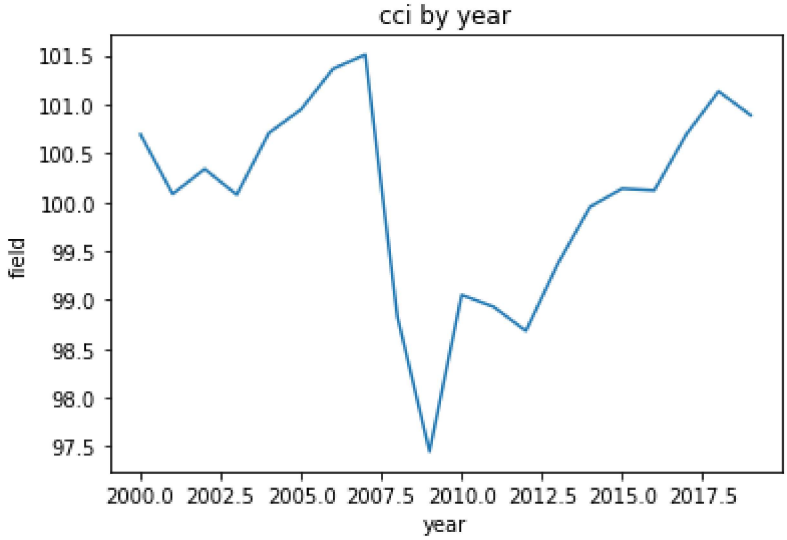
In [5]:
```
import matplotlib.pyplot as plt
import seaborn as sns

def plotvsyear(field):
    if field == 'year':
        return
    plt.plot(df['year'],df[field])
    plt.xlabel('year')
    plt.ylabel('field')
    plt.title('{0} by year'.format(field))
    plt.show()
    return
```

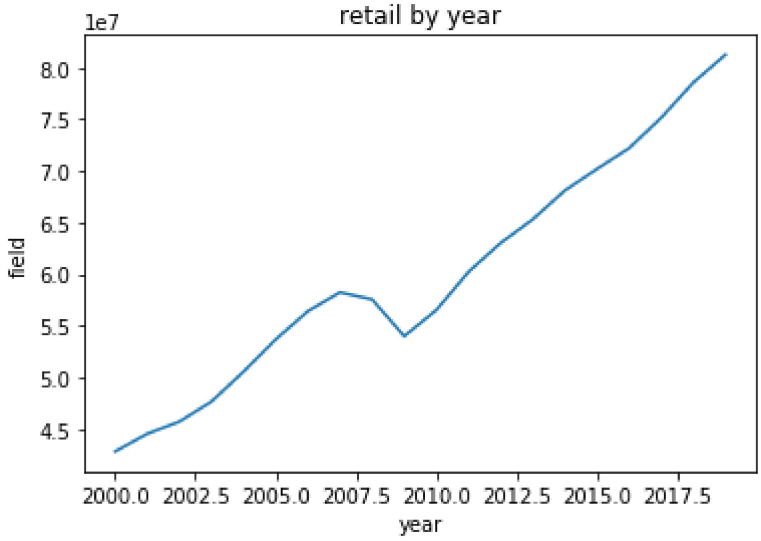
Create graphs (measure v year):

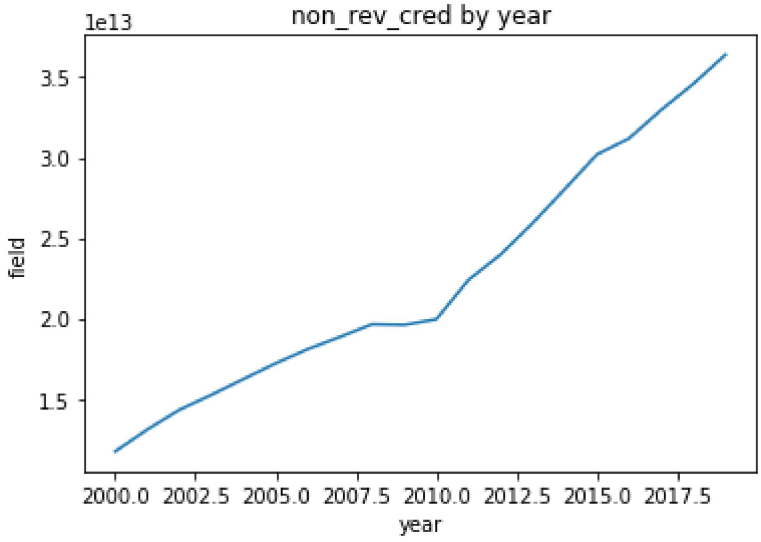
In [6]:
```
#plot by year
for field in df.columns:
    plotvsyear(field)
```

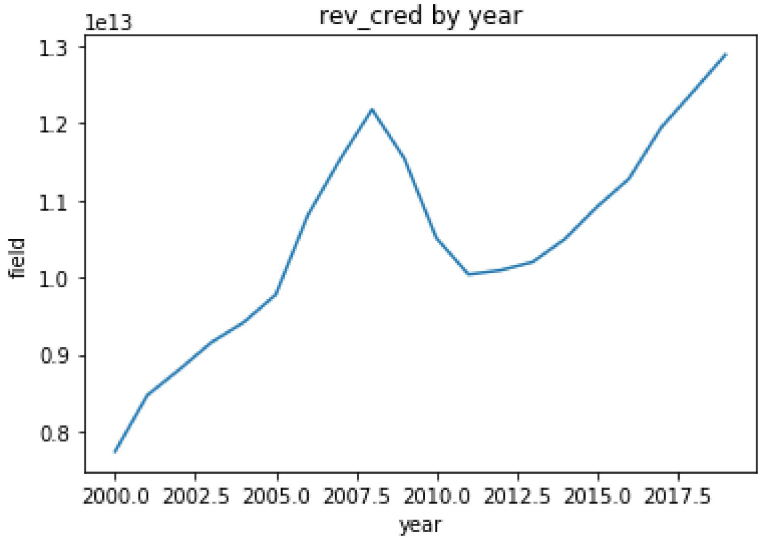

gdp by year
19000
18000
17000
16000
15000
14000
13000
field
2000.0 2002.5 2005.0 2007.5 2010.0 2012.5 2015.0 2017.5
year

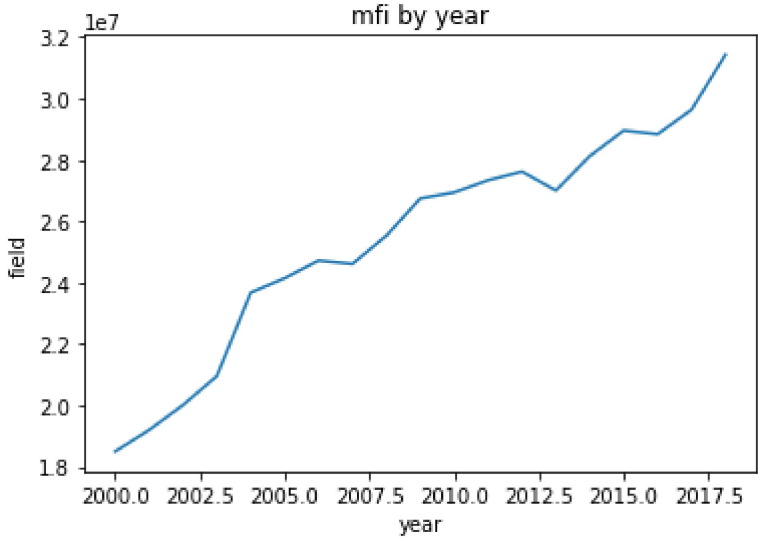

g17 by year
92
91
90
89
88
87
86
field
2011 2012 2013 2014 2015 2016 2017 2018 2019
year


cci by year
101.5
101.0
100.5
100.0
99.5
99.0
98.5
98.0
97.5
field
2000.0 2002.5 2005.0 2007.5 2010.0 2012.5 2015.0 2017.5
year


retail by year
1e7
8.0
7.5
7.0
6.5
6.0
5.5
5.0
4.5
field
2000.0 2002.5 2005.0 2007.5 2010.0 2012.5 2015.0 2017.5
year


non_rev_cred by year
1e13
3.5
3.0
2.5
2.0
1.5
field
2000.0 2002.5 2005.0 2007.5 2010.0 2012.5 2015.0 2017.5
year


rev_cred by year
1e13
1.3
1.2
1.1
1.0
0.9
0.8
field
2000.0 2002.5 2005.0 2007.5 2010.0 2012.5 2015.0 2017.5
year



Since the scaling is off, it will make it hard to visualize the differences between measures so I'll take the z-score to normalize.

Create function to generate z-score:

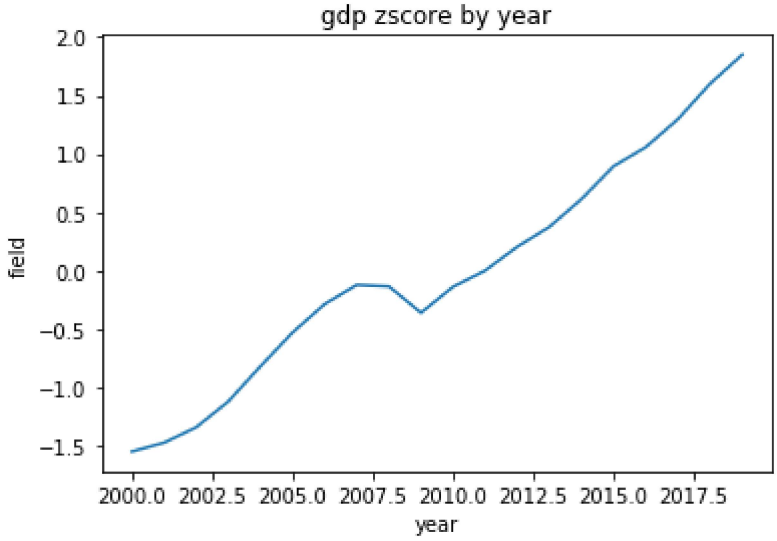
In [7]:
```
def zscores(field):
    if field == 'year':
        return
    mean = df[field].mean()
    std = df[field].std()
    xvalues = df[field].tolist()
    zscore = []
    for x in xvalues:
        zscore.append((x-mean)/std)
    new_field = str(field) + ' zscore'
    df[new_field] = zscore
    return
```

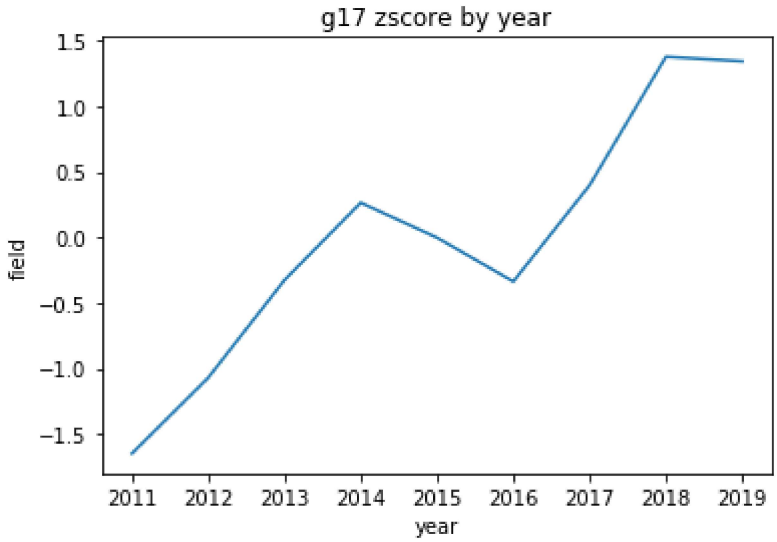
Generate z-scores:

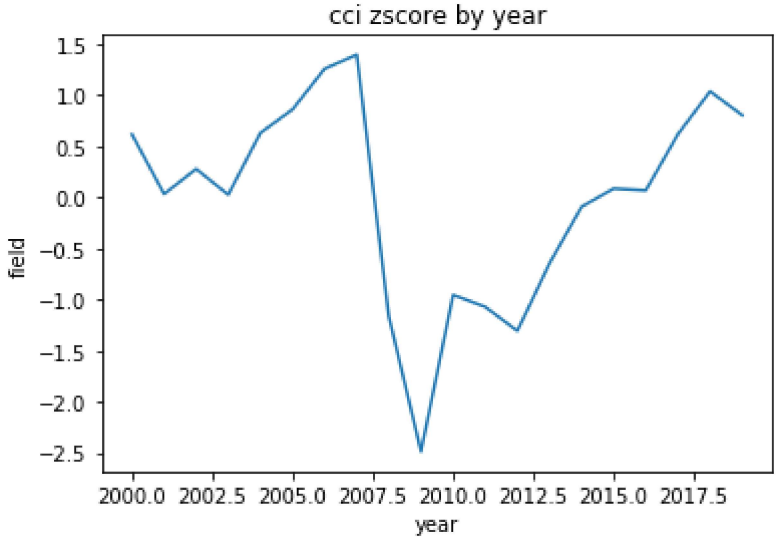
In [8]:
```
#generate zscores
for field in df.columns:
    zscores(field)
```

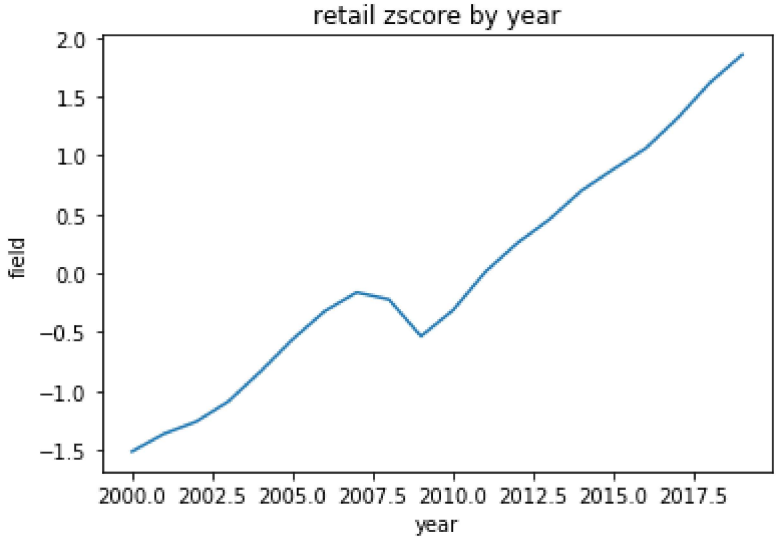
Plot z-scores by year:

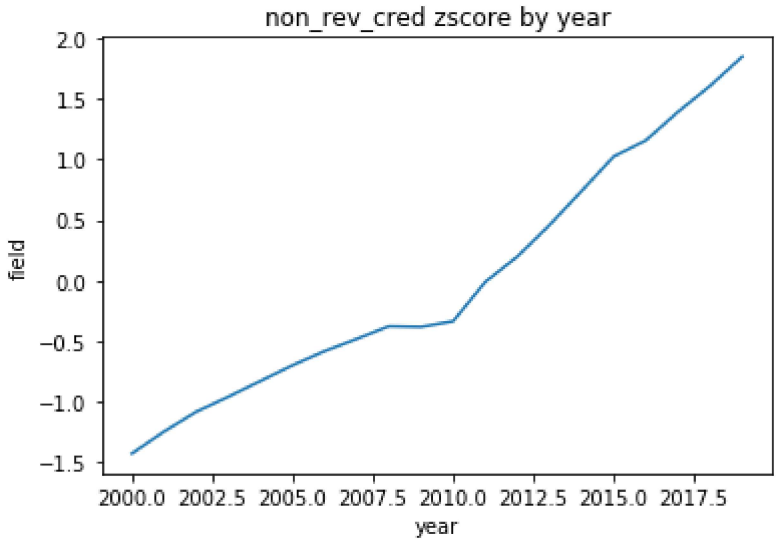
In [9]:
```
#plot zscores by year
for field in df.columns:
    if field.find('zscore') != -1:
        plotvsyear(field)
```

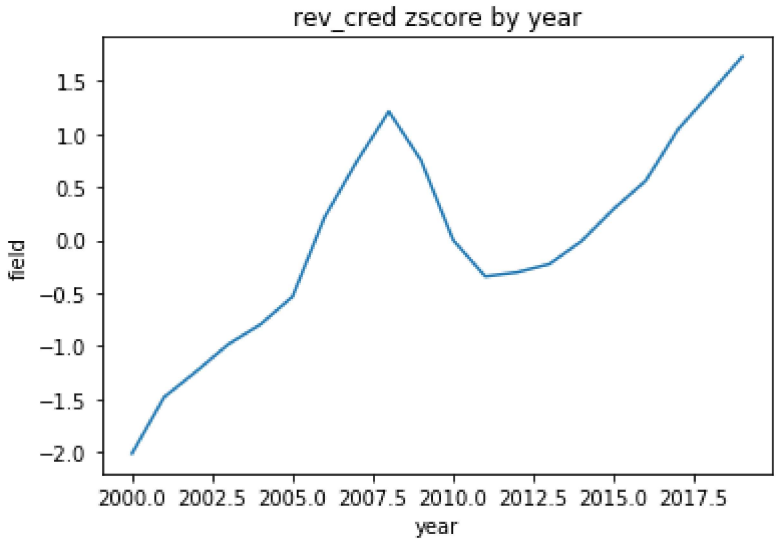

gdp zscore by year
field
year
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
2000.0
2002.5
2005.0
2007.5
2010.0
2012.5
2015.0
2017.5

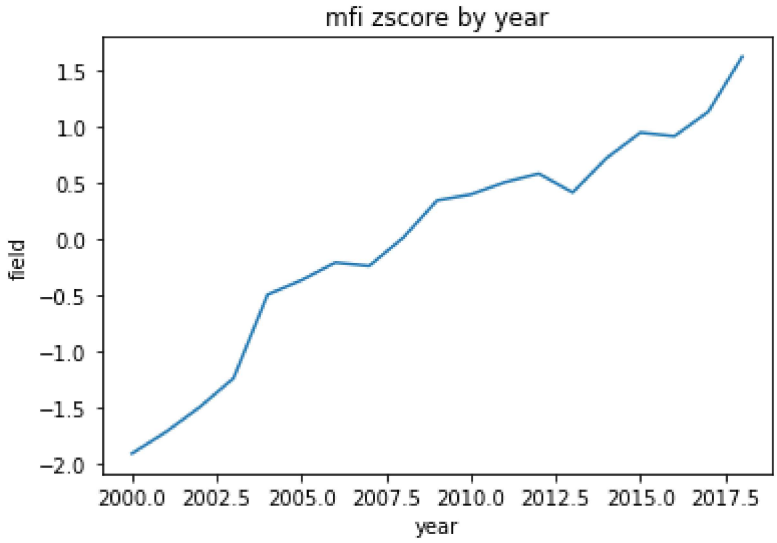

g17 zscore by year
field
year
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
2011
2012
2013
2014
2015
2016
2017
2018
2019


cci zscore by year
field
year
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
−2.5
2000.0
2002.5
2005.0
2007.5
2010.0
2012.5
2015.0
2017.5


retail zscore by year
field
year
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
2000.0 2002.5 2005.0 2007.5 2010.0 2012.5 2015.0 2017.5


non_rev_cred zscore by year
field
year
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
2000.0 2002.5 2005.0 2007.5 2010.0 2012.5 2015.0 2017.5


rev_cred zscore by year
field
year
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
2000.0 2002.5 2005.0 2007.5 2010.0 2012.5 2015.0 2017.5



Each of these graphs show a general upward trajectory over time. While not every measure shows the same information, there are some consistent themes that are presented. One of the themes is a major dip in 2008, which we know 2008 had a major recession. Our retail transactions, consumer confidence, revolving credit, and GDP all present the same major dip in 2008-2009.

To check how closely these measurements are related, we can evaluate the correlation coefficients for each relationship.

Check the correlations between measures:

In [10]:
```
zscores = [x for x in df.columns if x.find('zscore') != -1]

#correlations
corr = df[zscores].corr(method='pearson')
corr
```

Out[10]:

	gdp zscore	g17 zscore	cci zscore	retail zscore	non_rev_cred zscore	rev_cred zscore	mfi zscore
gdp zscore	1.000000	0.918829	0.095377	0.997046	0.985192	0.837321	0.958801
g17 zscore	0.918829	1.000000	0.927307	0.933867	0.922410	0.877054	0.832536
cci zscore	0.095377	0.927307	1.000000	0.122321	0.079532	-0.007990	-0.179348
retail zscore	0.997046	0.933867	0.122321	1.000000	0.990911	0.806202	0.940031
non_rev_cred zscore	0.985192	0.922410	0.079532	0.990911	1.000000	0.763318	0.921576
rev_cred zscore	0.837321	0.877054	-0.007990	0.806202	0.763318	1.000000	0.820334
mfi zscore	0.958801	0.832536	-0.179348	0.940031	0.921576	0.820334	1.000000

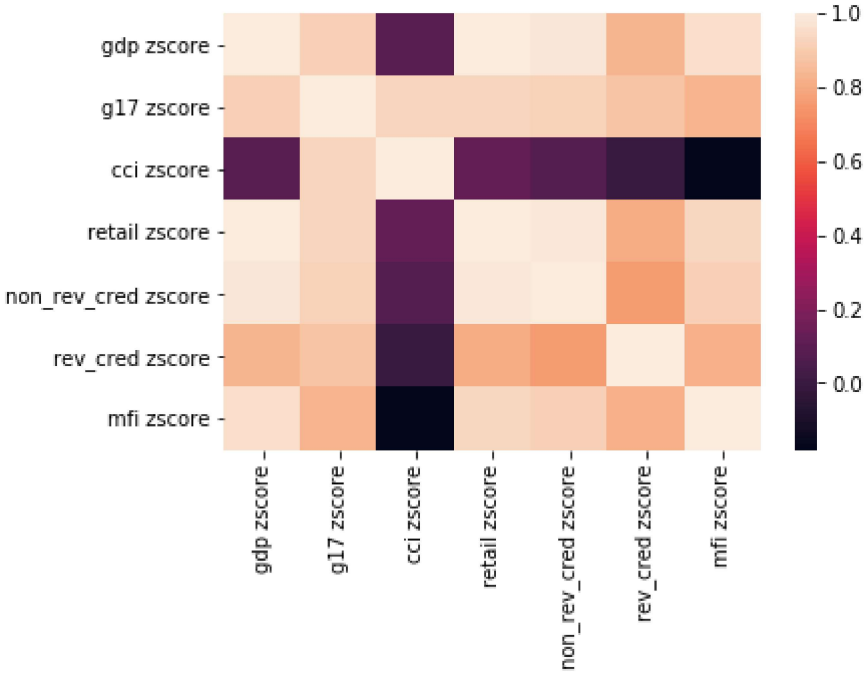
As expected, most of these measurements are highly correlated to GDP. Revolving credit and consumer confidence actually exhibit the least correlation to GDP.

Let's look at a heatmap of the correlation.

Generate heatmap of correlation:

In [11]:
```
sns.heatmap(corr)
```

Out[11]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x2636ce6d0f0>
```



I think the insight I would take away this data is that GDP, industrial utilization (g17), retail transactions, non-revolving credit, and median family income are essentially all measuring the same thing.

Interestingly, consumer confidence is not very correlated to retail transactions. The traditional assumption is that people buy less if they're less confident in the direction of the economy, but this seems to indicate otherwise. More detail on this relationship is needed however, because retail transactions encompass a very wide range of categories and this may certainly be true for specific categories. The interesting thing is that if that is the case, retail transactions appear to self-correct to match GDP.

Consumer confidence actually appears to be most related to the industrial utilization (g17). This is notable because g17 is highly correlated to GDP and most of our other measurements. This may indicate that consumer confidence is a reactive measurement that follows the effects of the general direction of the economy. To see an example of this we can look at the most notable movement during the time period, the 2008 recession. Immediately prior to the recession consumer confidence was at an all time high, and following the recession it dropped further than any other measurement. This would indicate that consumer confidence is a highly volitile reactive measurement.

Similarly to consumer confidence, revolving credit seems to have the strongest relationship to g17. Further review of the graph seems to indicate that it is also a reative measurement, while a highly less volitile one than consumer confidence. This indicates that following negative economic events, consumers and credit issuers opt to maintain lower credit exposure than usual.

Unfortunately, since most of the measurements actually present the same information and the others are reactive measurements, the best measurement we can use to predict future economic growth given this data appears to be GDP.

To do this, I will use linear regression to model potential future economic growth.

Linear regression models the relationship between dependent variables and an independent variable. For this case, I will use simple linear regression as there is only one viable dependent variable.

dependent: year independent: gdp

Create simple linear regression to predict gdp:

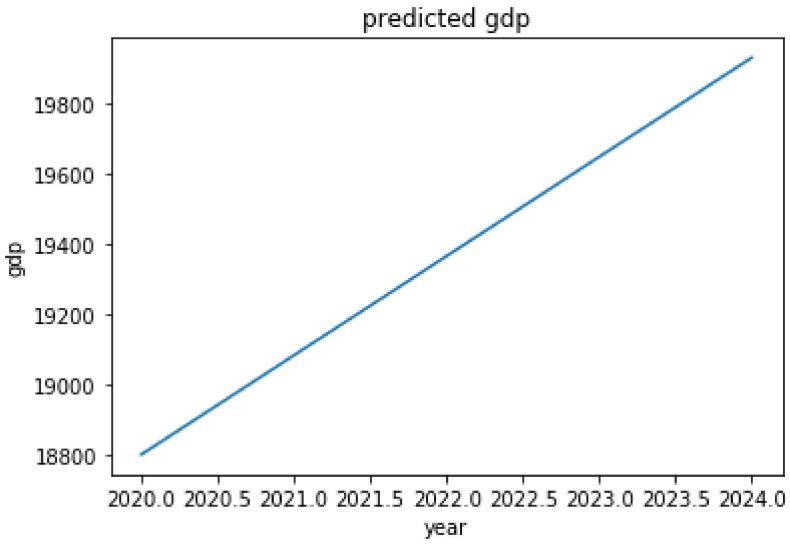
In [12]:
```
#linear regression
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split

y = df['gdp'].values.reshape(-1,1)
X = df['year'].values.reshape(-1,1)

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.1, rando
m_state=0)
reg = LinearRegression()
gdp_pred = reg.fit(X_train,y_train)
results = gdp_pred.predict(X_test)
print('''
actual: {0}
predict: {1}'''.format(y_test,results))
```

```
actual: [[18638.2]
 [13262.1]]
predict: [[18234.73941723]
 [13422.73836055]]
```

The model appears to closely match the actual results, so I'll extrapolate the model for the next five years.

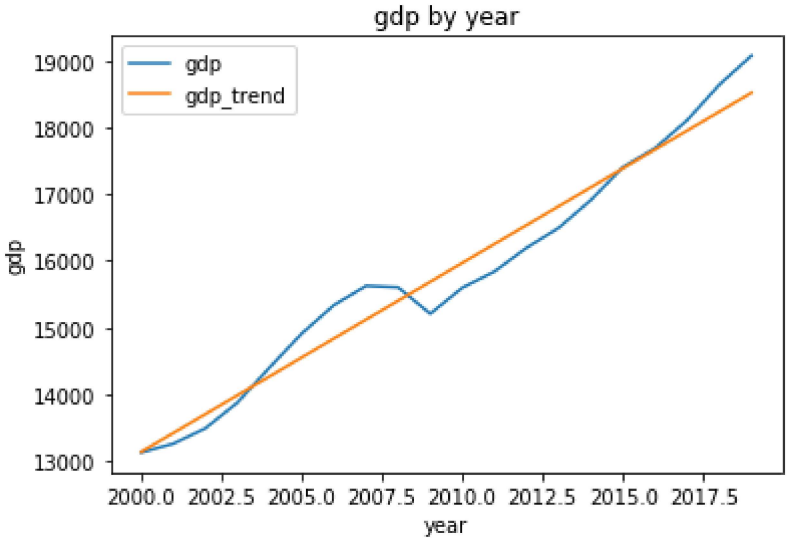
Predict gdp for the next five years:

In [13]:
```
#next five years gdp
next_five_years = pd.Series([2020,2021,2022,2023,2024]).values.reshape(-1,1)
n5 = gdp_pred.predict(next_five_years)
plt.plot(next_five_years,n5)
plt.xlabel('year')
plt.ylabel('gdp')
plt.title('predicted gdp')
plt.show()
```



Check trend vs actual:

In [14]:
```
#trend vs actual
trend = gdp_pred.predict(X)
df['gdp_trend'] = trend
plt.plot(df['year'],df[['gdp','gdp_trend']])
plt.xlabel('year')
plt.ylabel('gdp')
plt.title('gdp by year')
plt.legend(['gdp','gdp_trend'])
plt.show()
```



Unfortunately given the small sample in this example, the regression model will be prone to overfitting and this appears to be the case here. This model uses ordinary least squares to estimate the unknown parameters. OLS minimizes the sum of the squares of the differences between the dependent variable and those predicted by the linear function.

There are a few things we can try to combat this potential overfitting problem:

- The first is the selection of the training data. In this case, the first year and last year were chosen as the test parameters. This means that the events preceding and following the recession were likely given too much weight in the model. To combat this we can use cross validation to test the partitioning of the data.
- We can also try a few different estimation methods other than OLS, such as Lasso, Ridge, or Elastic regression

Leave One Out Cross Validation (LOO):

LOO splits the data set into samples that only leaves one record as the test set and use the remaining as the training set. This forces every record to take a turn as the test set and operate as part of the training set n-1 times.

In [74]:
```
from sklearn.model_selection import LeaveOneOut
from sklearn.model_selection import cross_validate

loo = LeaveOneOut()
loo.get_n_splits(X)

for train, test in loo.split(X,y):
    cvm = cross_validate(reg,X[train],y[train], cv=5)
    print(cvm['test_score'])
    model = reg.fit(X[train],y[train])
    res = model.predict(X)
    pred_df = pd.DataFrame()
    pred_df['year'] = df['year'].tolist()
    pred_df['gdp'] = df['gdp'].tolist()
    pred_df['pred'] = res
    plt.plot(pred_df['year'],pred_df[['gdp','pred']])
    plt.xlabel('year')
    plt.ylabel('gdp')
    plt.title('gdp by year')
    plt.legend(['gdp','pred'])
    plt.show()
    pred_df = None
```

```
[ 0.83323339 -3.29135586 -1.18587701  0.6119744  -1.47199289]
```


```
[ 0.89870647 -2.88992157 -1.26246949  0.62097986 -1.59736522]
```


```
[ 0.93510854 -2.7485686  -1.29331946  0.6185123  -1.5685429 ]
```

```
[ 0.89245704 -2.89015398 -1.24961062  0.61517311 -1.43703894]
```


```
[ 0.64017801 -3.23712436 -1.13709402  0.62338626 -1.31696408]
```


```
[ 0.82908022 -0.22753466 -1.0339299   0.64220943 -1.31091746]
```

```
[ 0.87120027 -0.01387257 -0.96738543  0.66331283 -1.39509459]
```


```
[ 0.8418811  -0.31143026 -0.9655652   0.67399103 -1.50621524]
```


```
[ 0.6353716  -0.67296622 -1.12083182  0.64048743 -1.47795503]
```

```
[ 0.10002314 -0.36523094 -2.23189383  0.51239832 -1.09365246]
```


```
[ 0.28001587 -0.41270123 -0.68636709  0.51721754 -1.11438529]
```


```
[ 0.33535213 -0.40610581 -0.64098915  0.48603843 -1.03049773]
```

```
[ 0.42889704 -0.43541975 -3.28969774  0.48647125 -1.07243734]
```


```
[ 0.48736716 -0.44687872 -3.29361298  0.66547453 -1.0659001 ]
```


```
[ 0.53308569 -0.48109826 -3.1729006   0.65485384 -1.27704013]
```

```
[ 0.52701825 -0.51683752 -2.96896999  0.49912391 -1.7566651 ]
```


```
[ 0.52940736 -0.51466838 -2.98021299  0.24296414 -1.87101666]
```


```
[ 0.46074568 -0.52155308 -2.84239757  0.30377046 -0.23483429]
```

```
[ 0.22867023 -0.51776153 -2.57324931  0.43512481  0.05957878]
```


```
[-0.08244792 -0.49455327 -2.39815317  0.52215064 -0.51364263]
```

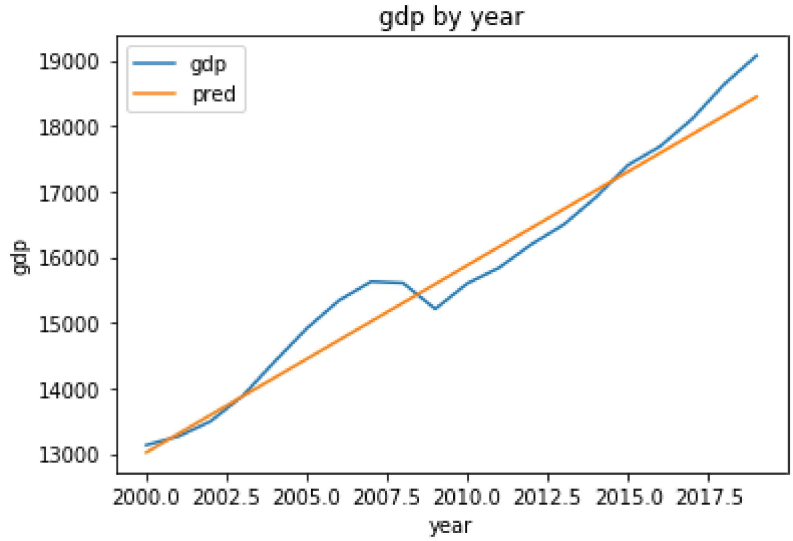
Repeated K-Fold:

K-Fold splits data into training and test sets by splitting the data into k consecutive folds. Repeated K-Fold repeats K-Fold n times while randomizing each repetition.

In [72]:
```
from sklearn.model_selection import RepeatedKFold
rkf = RepeatedKFold(n_splits=3, n_repeats=2)
for train, test in rkf.split(X,y):
    cvm = cross_validate(reg,X[train],y[train], cv=5)
    print(cvm['test_score'])
    model = reg.fit(X[train],y[train])
    res = model.predict(X)
    pred_df = pd.DataFrame()
    pred_df['year'] = df['year'].tolist()
    pred_df['gdp'] = df['gdp'].tolist()
    pred_df['pred'] = res
    plt.plot(pred_df['year'],pred_df[['gdp','pred']])
    plt.xlabel('year')
    plt.ylabel('gdp')
    plt.title('gdp by year')
    plt.legend(['gdp','pred'])
    plt.show()
    pred_df = None
```

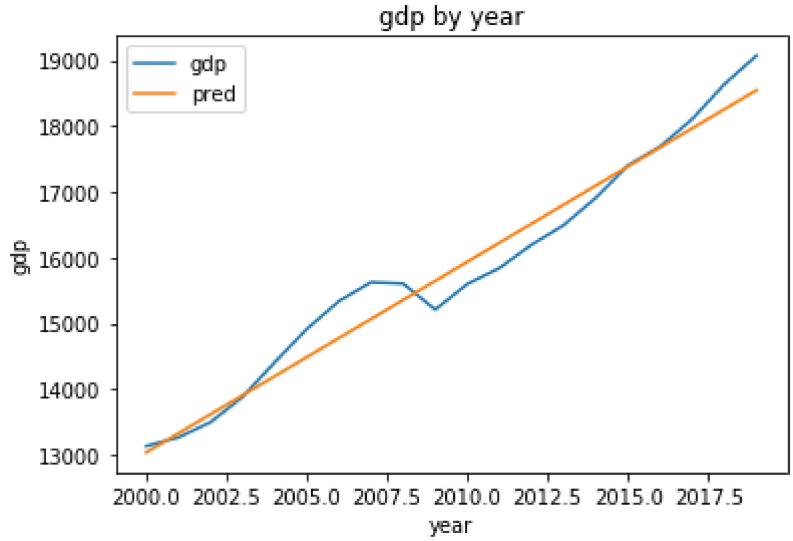
```
[  0.84321425 -10.27053319   0.38017006   0.48401438  -3.96962452]
```


```
[ 0.76112901 -0.0260068  -1.77055127 -3.5661225  -1.25995751]
```



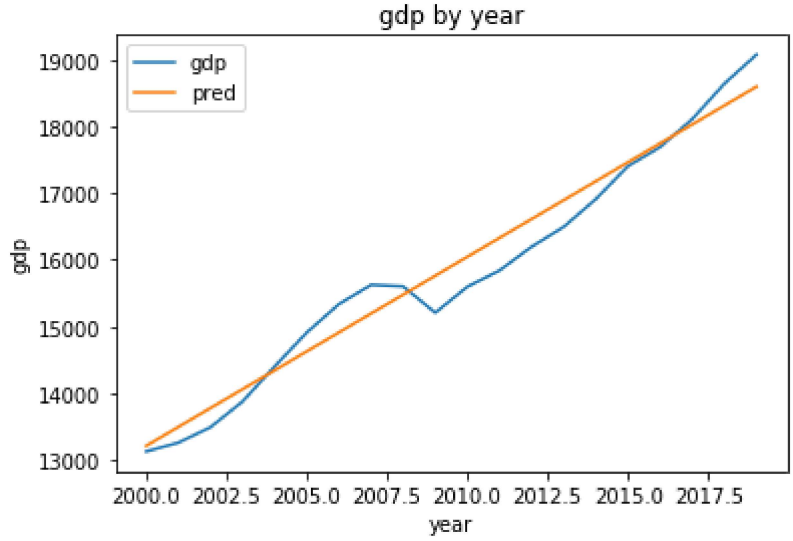

```
[ 0.82250405 -0.28841462 -0.55598732  0.3348465  -0.87324416]
```

```
[ 0.47817287  0.49834614 -1.44338323 -7.04803325  0.21960096]
```




```
[-0.63318042 -3.18922115  0.62588783  0.91986308 -6.8729261 ]
```


```
[-8.18322465e-01 -6.09424845e-02 -1.00400828e+03  2.70990048e-01
 -1.43856738e+00]
```

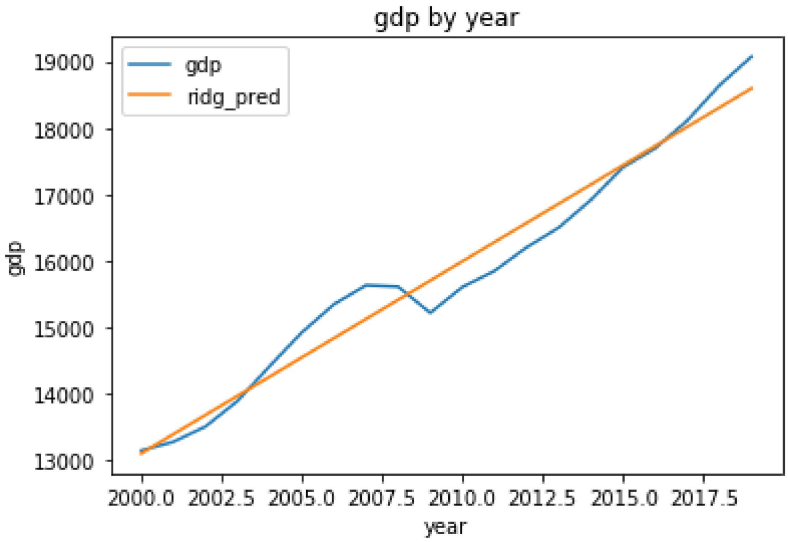


Based on the cross validation results, it looks like the training selection was not an issue. The cross validation test results look identical to the initial train/test split results.

Ridge Regression:

Ridge regression attempts to introduce bias into the model to correct multicollinearity or overfitting. This method is typically beneficial when there is a large number of parameters, which we don't have in this case.

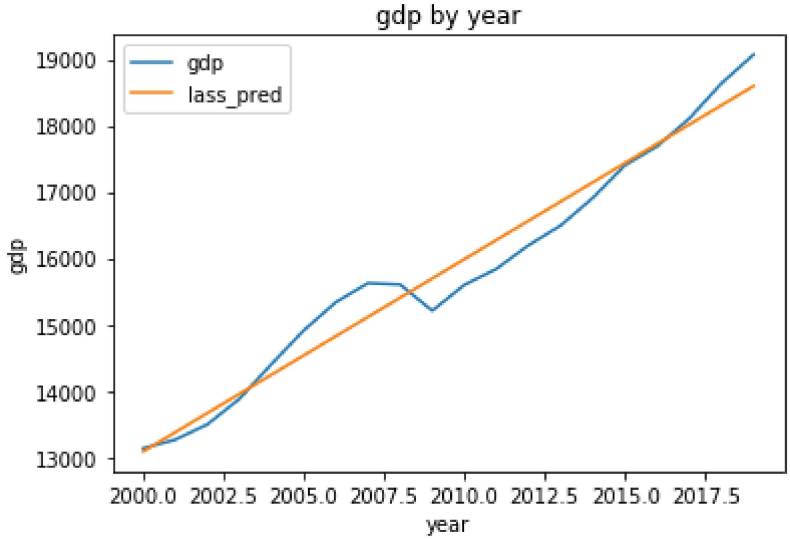
In [88]:
```
from sklearn.linear_model import Ridge
ridge = Ridge(alpha=1.0)
ridge.fit(X, y)
ridge_pred = ridge.predict(X)

ridge_df = pd.DataFrame()
ridge_df['year'] = df['year'].tolist()
ridge_df['gdp'] = df['gdp'].tolist()
ridge_df['ridg_pred'] = ridge_pred

plt.plot(ridge_df['year'],ridge_df[['gdp','ridg_pred']])
plt.xlabel('year')
plt.ylabel('gdp')
plt.title('gdp by year')
plt.legend(['gdp','ridg_pred'])
plt.show()
```



Lasso Regression:

Lasso regression performs covariate selection to simplify models and penalizes coefficients to improve model accuracy. It is unlikley that Lasso will provide any benefit in this case.

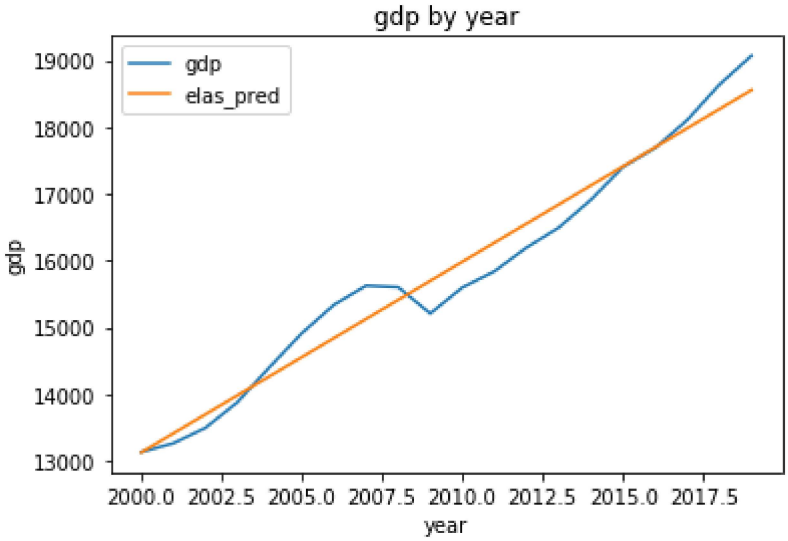
In [89]:
```
from sklearn.linear_model import Lasso
lasso = Lasso(alpha=1.0)
lasso.fit(X, y)
lasso_pred = lasso.predict(X)

lasso_df = pd.DataFrame()
lasso_df['year'] = df['year'].tolist()
lasso_df['gdp'] = df['gdp'].tolist()
lasso_df['lass_pred'] = lasso_pred

plt.plot(lasso_df['year'],lasso_df[['gdp','lass_pred']])
plt.xlabel('year')
plt.ylabel('gdp')
plt.title('gdp by year')
plt.legend(['gdp','lass_pred'])
plt.show()
```



ElasticNet Regression:

Elastic net regularization combines the penalties of the lasso and ridge regressions to overcome the weaknesses of each method. Given that neither lasso nor ridge regression provided much benefit, elastic net regularization will most likely not either.

In [90]:
```
from sklearn.linear_model import ElasticNet
elastic = ElasticNet(alpha=1.0)
elastic.fit(X, y)
elastic_pred = elastic.predict(X)

elastic_df = pd.DataFrame()
elastic_df['year'] = df['year'].tolist()
elastic_df['gdp'] = df['gdp'].tolist()
elastic_df['elas_pred'] = elastic_pred

plt.plot(elastic_df['year'],elastic_df[['gdp','elas_pred']])
plt.xlabel('year')
plt.ylabel('gdp')
plt.title('gdp by year')
plt.legend(['gdp','elas_pred'])
plt.show()
```



Review all results:

In [92]:
```
pred_df = pd.merge(elastic_df, lasso_df[['year','lass_pred']], how='inner', on
=None, left_on='year', right_on='year',
         left_index=False, right_index=False, sort=True,
         suffixes=('_x', '_y'), copy=True, indicator=False,
         validate=None)
pred_df = pd.merge(pred_df, ridge_df[['year','ridg_pred']], how='inner', on=No
ne, left_on='year', right_on='year',
         left_index=False, right_index=False, sort=True,
         suffixes=('_x', '_y'), copy=True, indicator=False,
         validate=None)
pred_df = pd.merge(pred_df, df[['year','gdp_trend']], how='inner', on=None, le
ft_on='year', right_on='year',
         left_index=False, right_index=False, sort=True,
         suffixes=('_x', '_y'), copy=True, indicator=False,
         validate=None)
pred_df
```

Out[92]:

	year	gdp	elas_pred	lass_pred	ridg_pred	gdp_trend
0	2000	13131.0	13124.316296	13083.608571	13087.463333	13139.679475
1	2001	13262.1	13410.270370	13373.847669	13377.296667	13422.738361
2	2002	13493.1	13696.224444	13664.086767	13667.130000	13705.797246
3	2003	13879.1	13982.178519	13954.325865	13956.963333	13988.856132
4	2004	14406.4	14268.132593	14244.564962	14246.796667	14271.915018
5	2005	14912.5	14554.086667	14534.804060	14536.630000	14554.973903
6	2006	15338.3	14840.040741	14825.043158	14826.463333	14838.032789
7	2007	15626.0	15125.994815	15115.282256	15116.296667	15121.091675
8	2008	15604.7	15411.948889	15405.521353	15406.130000	15404.150560
9	2009	15208.8	15697.902963	15695.760451	15695.963333	15687.209446
10	2010	15598.8	15983.857037	15985.999549	15985.796667	15970.268332
11	2011	15840.7	16269.811111	16276.238647	16275.630000	16253.327217
12	2012	16197.0	16555.765185	16566.477744	16565.463333	16536.386103
13	2013	16495.4	16841.719259	16856.716842	16855.296667	16819.444989
14	2014	16912.0	17127.673333	17146.955940	17145.130000	17102.503874
15	2015	17403.8	17413.627407	17437.195038	17434.963333	17385.562760
16	2016	17688.9	17699.581481	17727.434135	17724.796667	17668.621646
17	2017	18108.1	17985.535556	18017.673233	18014.630000	17951.680532
18	2018	18638.2	18271.489630	18307.912331	18304.463333	18234.739417
19	2019	19072.7	18557.443704	18598.151429	18594.296667	18517.798303

Sum:

In [97]:
```
pred_df[[x for x in pred_df.columns if x != 'year']].sum()
```

Out[97]:
```
gdp          316817.600000
elas_pred    316817.600000
lass_pred    316817.600000
ridg_pred    316817.600000
gdp_trend    316574.777778
dtype: float64
```

Mean:

In [98]:
```
pred_df[[x for x in pred_df.columns if x != 'year']].mean()
```

Out[98]:
```
gdp          15840.880000
elas_pred    15840.880000
lass_pred    15840.880000
ridg_pred    15840.880000
gdp_trend    15828.738889
dtype: float64
```

Standard Deviation:

In [99]:
```
pred_df[[x for x in pred_df.columns if x != 'year']].std()
```

Out[99]:
```
gdp          1748.034352
elas_pred    1691.727117
lass_pred    1717.077658
ridg_pred    1714.677124
gdp_trend    1674.598951
dtype: float64
```

In [104]:
```
pred_df['elas_err'] = pred_df['gdp']-pred_df['elas_pred']
pred_df['lass_err'] = pred_df['gdp']-pred_df['lass_pred']
pred_df['ridg_err'] = pred_df['gdp']-pred_df['ridg_pred']
pred_df['slre_err'] = pred_df['gdp']-pred_df['gdp_trend']

print('Error Sum')
print(pred_df[[x for x in pred_df.columns if x.find('err') != -1]].sum())
print('''
Error Mean:''')
print(pred_df[[x for x in pred_df.columns if x.find('err') != -1]].mean())
print('''
Error Standard Dev:''')
print(pred_df[[x for x in pred_df.columns if x.find('err') != -1]].std())
```

```
Error Sum
elas_err    9.567884e-10
lass_err    9.567884e-10
ridg_err   -9.058567e-10
slre_err    2.428222e+02
dtype: float64

Error Mean:
elas_err    4.783942e-11
lass_err    4.783942e-11
ridg_err   -4.529284e-11
slre_err    1.214111e+01
dtype: float64

Error Standard Dev:
elas_err    327.580607
lass_err    326.584419
ridg_err    326.594549
slre_err    329.358376
dtype: float64
```

Summary: Each of the different regression and train data selection methods appear to present similar results. Given that the only feature is year and this is a small subset of summary data, this limits the potential benefits that can be derived from tuning/changing models as well as insights that can be derived. Unfortunately, the measures and data that were colected seem to be unable to effectively predict economic health. However, time seems to do a generally decent job at predicting GDP growth (at least in the short term).

In order to effectively predict future economic health/growth, the data selection will likely have to extend beyond the traditional measures or possibly take a more detailed look into the specific industry/market issues that lead to extraodinary economic events (the housing collapse in 2008 for example).