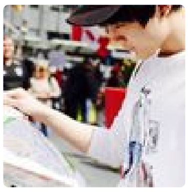
 

Sean Choi

BlackFriday_EDA_MLfitting

last run a day ago · IPython Notebook HTML · 82 views
using data from Black Friday · Public

Notebook

In [1]:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
sns.set(style="whitegrid")
%matplotlib inline
```

In [2]:

```
from sklearn.model_selection import train_test_split
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import learning_curve
from sklearn.model_selection import cross_val_score
from sklearn.preprocessing import LabelEncoder
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_squared_error
```

Background

BlackFriday dataset was downloaded from Keggle (Mehdi Dagdoug). The dataset contains samples of transactions made in a retail store on Black Friday. Purpose of this project is to define a model that can predict ***Purchase*** given new customer's information.

In [3]:

```
df = pd.read_csv('../input/BlackFriday.csv')
```

Quick overview of the data

This dataset has 537577 entries with 12 columns. It is a lot of data (~half million) that contains customer-specific and store-specific information.

Dataset is mostly clean as it is, but some cleaning is still necessary about their dtype and NaNs.

- Most features are categorical
- A few columns contain lots of NaNs

In [4]:

```
print(df.info())
```

```
<class 'pandas.core.frame.DataFrame'>
```

```
RangeIndex: 537577 entries, 0 to 537576
Data columns (total 12 columns):
User_ID                       537577 non-
null int64
Product_ID                    537577 non-
null object
Gender                        537577 non-
null object
Age                           537577 non-
null object
Occupation                    537577 non-
null int64
City_Category                 537577 non-
null object
Stay_In_Current_City_Years    537577 non-
null object
Marital_Status                537577 non-
null int64
Product_Category_1            537577 non-
null int64
Product_Category_2            370591 non-
null float64
Product_Category_3            164278 non-
null float64
Purchase                      537577 non-
null int64
dtypes: float64(2), int64(5), object(5)
memory usage: 49.2+ MB
None
```

A brief investigation reveals that ~31% and ~70% data are NaNs in ***Product_Category_2*** and ***Product_Categroy_3***, respectively.

In [5]:

```
print(df.isnull().any())
missing_ser_percentage = (df.isnull().sum()/df.shape[
0]*100).sort_values(ascending=False)
missing_ser_percentage = missing_ser_percentage[missi
ng_ser_percentage!=0].round(2)
missing_ser_percentage.name = 'missing values %'
print('\nNaN ratio')
print(missing_ser_percentage)
```

```
User_ID                       False
Product_ID                    False
Gender                        False
Age                           False
Occupation                    False
City_Category                 False
```

```
City_Category                 False
Stay_In_Current_City_Years    False
Marital_Status                False
Product_Category_1            False
Product_Category_2             True
Product_Category_3             True
Purchase                      False
dtype: bool

NaN ratio
Product_Category_3    69.44
Product_Category_2    31.06
Name: missing values %, dtype: float64
```

A number of unique element in each column:

In [6]:

```
for col in df.columns:
    print('{} unique element: {}'.format(col,df[col].
nunique()))
```

```
User_ID unique element: 5891
Product_ID unique element: 3623
Gender unique element: 2
Age unique element: 7
Occupation unique element: 21
City_Category unique element: 3
Stay_In_Current_City_Years unique elemen
t: 5
Marital_Status unique element: 2
Product_Category_1 unique element: 18
Product_Category_2 unique element: 17
Product_Category_3 unique element: 15
Purchase unique element: 17959
```

In order to utilize ***Product_Category_2*** and ***Product_Category_3***, fill their NaNs with 0.

In [7]:

```
df.fillna(0,inplace=True)
```

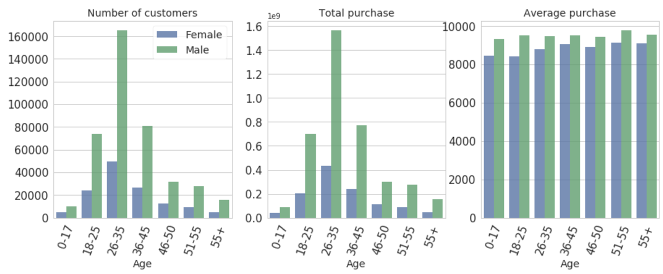
Exploratory data analysis

While investigating all different features at once is difficult, stuyding groups of two/three features separately may reveal important stories about the dataset.

Count/Purchase by Age by Gender

In [8]:

```
age_order = ['0-17','18-25','26-35','36-45','46-50',
'51-55','55+']
plt.figure(figsize=(15,5))
plt.subplot(131)
sns.countplot('Age',order=age_order,hue='Gender',data
=df,alpha = 0.8)
plt.xlabel('Age',fontsize=14)
plt.ylabel('')
plt.xticks(rotation=70)
plt.title('Number of customers',fontsize=14)
plt.legend(['Female','Male'],frameon=True,fontsize=14
)
plt.tick_params(labelsize=15)
plt.subplot(132)
df_Tpurchase_by_Age = df.groupby(['Age','Gender']).ag
g({'Purchase':np.sum}).reset_index()
sns.barplot('Age','Purchase',hue='Gender',data=df_Tpu
rchase_by_Age,alpha = 0.8)
plt.xlabel('Age',fontsize=14)
plt.ylabel('')
plt.xticks(rotation=70)
plt.title('Total purchase',fontsize=14)
plt.legend().set_visible(False)
plt.tick_params(labelsize=15)
plt.subplot(133)
df_Apurchase_by_Age = df.groupby(['Age','Gender']).ag
g({'Purchase':np.mean}).reset_index()
sns.barplot('Age','Purchase',hue='Gender',data=df_Apu
rchase_by_Age,alpha = 0.8)
plt.xlabel('Age',fontsize=14)
plt.ylabel('')
plt.xticks(rotation=70)
plt.title('Average purchase',fontsize=14)
plt.legend().set_visible(False)
plt.tick_params(labelsize=15)
```



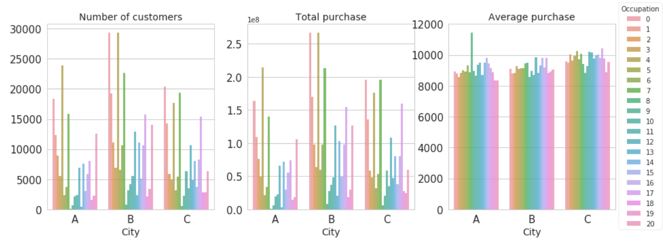
In both Gender, the Age group that purchased most was [26-35]. The total number of customers who made transactions and their total purchase amount were very strongly correlated. This is well reflected in the thrid subplot where the average purchase amount was very similiar in all Age groups. This may mean that ***Age*** may not be a strong predictor for ***Purchase***. However, ***Gender*** may be helpful as it was clear that Male spent mor than Female.

Assuming the Gender ratio to be simliar in each city, different city may have different occupation distribution. If that is the case, the ***city*** that the customer came from may be a strong predictor for ***Purchase*** as customers with certain occupations may spend more during shopping.

Count/Purchase by City by Occupation

In [9]:

```
city_order = ['A','B','C']
plt.figure(figsize=(15,5))
plt.subplot(131)
sns.countplot('City_Category',order=city_order,hue='Occupation',data=df,alpha = 0.8)
plt.xlabel('City',fontsize=14)
plt.ylabel('')
plt.legend().set_visible(False)
plt.tick_params(labelsize=15)
plt.title('Number of customers',fontsize=14)
plt.subplot(132)
df_Tpurchase_by_City = df.groupby(['City_Category','Occupation']).agg({'Purchase':np.sum}).reset_index()
sns.barplot('City_Category','Purchase',hue='Occupation',data=df_Tpurchase_by_City,alpha = 0.8)
plt.title('Total purchase',fontsize=14)
plt.xlabel('City',fontsize=14)
plt.ylabel('')
plt.legend().set_visible(False)
plt.tick_params(labelsize=15)
plt.subplot(133)
df_Apurchase_by_City = df.groupby(['City_Category','Occupation']).agg({'Purchase':np.mean}).reset_index()
sns.barplot('City_Category','Purchase',hue='Occupation',data=df_Apurchase_by_City,alpha = 0.8)
plt.title('Average purchase',fontsize=14)
plt.xlabel('City',fontsize=14)
plt.ylabel('')
plt.legend(title='Occupation',frameon=True,fontsize=10,bbox_to_anchor=(1,0.5), loc="center left")
plt.tick_params(labelsize=15)
```



However, in each City, the Occupation distribution was quite similar. As observed in the Age distribution above, the number of customers and total purchase amounts from customers in different occupations in each city showed strong correlation. This is also well reflected in the last subplot where the average purchase was similiar in all different groups of consideration. One thing to be noted is that the average purchase of customers with *Occupation #8* from *City A* showed distinctively high average purchase.

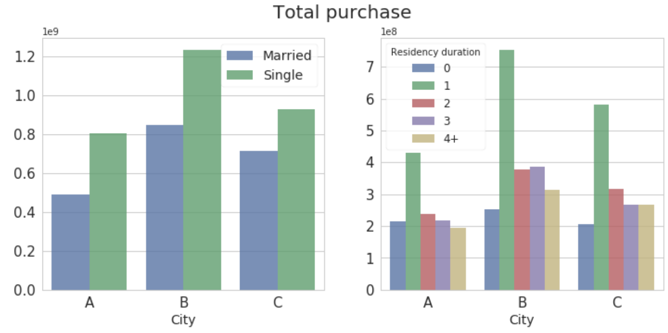
Other features can also be investigated in terms of ***Purchase***.

Purchase by City by Marrital status and Residency duration

In [10]:

```
df['Marital_Status_label']=np.where(df['Marital_Statu
s'] == 0,'Single','Married')
df_Tpurchase_by_City_Marital = df.groupby(['City_Cate
gory','Marital_Status_label']).agg({'Purchase':np.sum
}).reset_index()
df_Tpurchase_by_City_Stay = df.groupby(['City_Categor
y','Stay_In_Current_City_Years']).agg({'Purchase':np.
sum}).reset_index()
fig = plt.figure(figsize=(12,5))
fig.suptitle('Total purchase',fontsize=20)
plt.subplot(121)
sns.barplot('City_Category','Purchase',hue='Marital_S
tatus_label',data=df_Tpurchase_by_City_Marital,alpha
= 0.8)
plt.xlabel('City',fontsize=14)
plt.ylabel('')
plt.legend(frameon=True,fontsize=14)
plt.tick_params(labelsize=15)
plt.subplot(122)
sns.barplot('City_Category','Purchase',hue='Stay_In_C
urrent_City_Years',data=df_Tpurchase_by_City_Stay,alp
ha = 0.8)
plt.xlabel('City',fontsize=14)
plt.ylabel('')
plt.legend(title='Residency duration',frameon=True,fo
ntsize=12,loc=2)
```

```
plt.tick_params(labelsize=15)
```



It was shown that unmarried customers spent more money than the married. Customers who lived in their city for 1 year tent to spend more than other groups.

Also, it would be interesting to find the ***Product_Category_1*** that was the most famous.

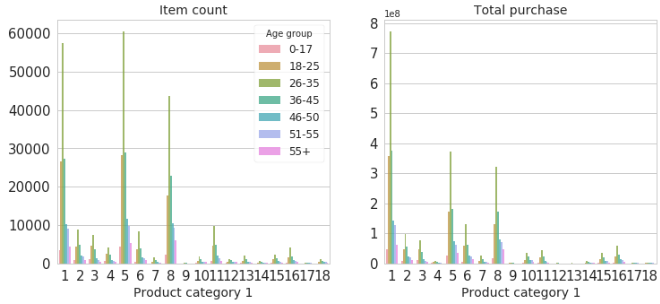
Count/Purchase by Product_Category_1 by Age and Gender

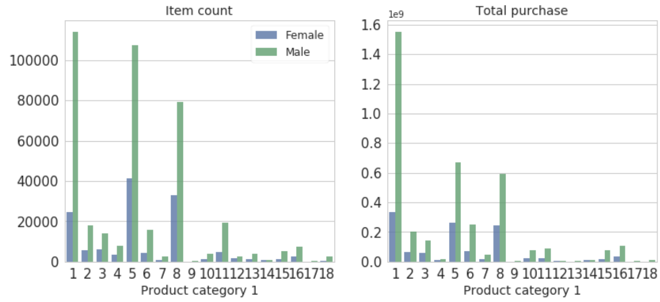
In [11]:

```
df_Tpurchase_by_PC1_Age = df.groupby(['Product_Catego
ry_1','Age']).agg({'Purchase':np.sum}).reset_index()
fig = plt.figure(figsize=(12,5))
plt.subplot(121)
sns.countplot('Product_Category_1',hue='Age',data=df,
alpha = 0.8,hue_order=age_order)
plt.title('Item count',fontsize=14)
plt.xlabel('Product category 1',fontsize=14)
plt.ylabel('')
plt.legend(title='Age group',frameon=True,fontsize=12
)
plt.tick_params(labelsize=15)
plt.subplot(122)
sns.barplot('Product_Category_1','Purchase',hue='Age'
,data=df_Tpurchase_by_PC1_Age,alpha = 0.8)
plt.title('Total purchase',fontsize=14)
plt.xlabel('Product category 1',fontsize=14)
plt.ylabel('')
plt.legend().set_visible(False)
plt.tick_params(labelsize=15)

df_Tpurchase_by_PC1_Gender = df.groupby(['Product_Cat
egory_1','Gender']).agg({'Purchase':np.sum}).reset_in
dex()
fig = plt.figure(figsize=(12,5))
plt.subplot(121)
sns.countplot('Product_Category_1',hue='Gender',data=
df,alpha = 0.8)
```

```
plt.title('Item count',fontsize=14)
plt.xlabel('Product category 1',fontsize=14)
plt.ylabel('')
plt.legend(['Female','Male'],frameon=True,fontsize=12
)
plt.tick_params(labelsize=15)
plt.subplot(122)
sns.barplot('Product_Category_1','Purchase',hue='Gend
er',data=df_Tpurchase_by_PC1_Gender,alpha = 0.8)
plt.title('Total purchase',fontsize=14)
plt.xlabel('Product category 1',fontsize=14)
plt.ylabel('')
plt.legend().set_visible(False)
plt.tick_params(labelsize=15)
```





In general,

- Male shopped more than Female
- Single shopped more than Married
- Customers from *City B* shopped the most
- Customers who has resided in their city for 1 year shopped the most
- *Product_category_1 #1,5,8* were the most selling
- *Product_category_1 #1* made the most profit

These relationships between different features can be investigated further to set the new marketing strategy to maximize the profit of the retail store (possibly for the future black Fridays). For example, the retail store may consider doing more advertisements targetting their unmarried male customers in *City B* on *product_category_1 #1*.

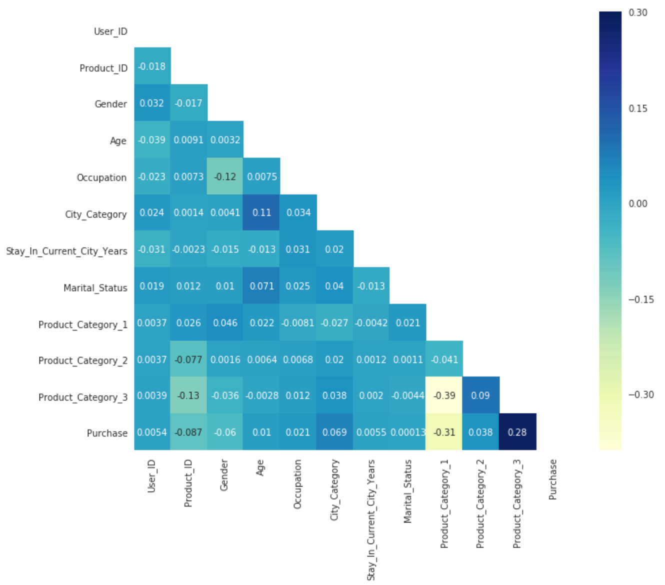
As the figures presented above only capture the relationships between maximum three different features, more general correlation plot can be considered.

In order to analyze the general correlation between different features, the categorical data need to be converted to **numerical counterparts**.

In [12]:

```
le_U_ID = LabelEncoder()
df['User_ID'] = le_U_ID.fit_transform(df['User_ID'])
le_P_ID = LabelEncoder()
df['Product_ID'] = le_P_ID.fit_transform(df['Product_
ID'])
df['Gender'] = np.where(df['Gender']=='F',1,0) # Mal
e: 0, Female: 1
Age_map = {'0-17':0,'26-35':1, '46-50':2, '36-45':3,
'18-25':4,'51-55':5,'55+':6}
df['Age'] = df['Age'].map(Age_map)
City_map = {'A':0,'B':1,'C':2}
df['City_Category'] = df['City_Category'].map(City_ma
p)
df['Stay_In_Current_City_Years'] = np.where(df['Stay_
In_Current_City_Years']=='4+','4',df['Stay_In_Current
_City_Years'])
df['Stay_In_Current_City_Years'] = df['Stay_In_Curren
t_City_Years'].astype(int)

corr = df.corr()
mask = np.zeros_like(corr, dtype=np.bool)
mask[np.triu_indices_from(mask)] = True
plt.figure(figsize=(11, 9))
ax = sns.heatmap(corr, mask=mask, square = True, vmax
 = 0.3,annot=True,cmap="YlGnBu")
sns.set(font_scale = 1.5)
df.drop('Marital_Status_label',axis=1,inplace=True)
```



ML model fitting

Dataset is separated into **X** and **y** as input features and output, respectively.

Model chosen: Random Forest (RF)

Since most features are not continuous, *Random Forest Regressor* is expected to fit the data well. Also, the fact that this dataset has ~10 features, RF is expected to perform reasonably well. Since the given dataset contains ~half million entries, using all of them may cause running-time issue on my machine when trying to do some iterative works like generating the learning curve. Therefore, only the fraction (1/50) of its data (~26k) will be randomly sampled for initial ML model fitting attempts.

In [13]:

```
df_frac = df.sample(frac=0.05)
X = df_frac.drop(['Purchase'], axis=1)
y = df_frac['Purchase']
X_train,X_test,y_train,y_test = train_test_split(X,y,
random_state=100)
```

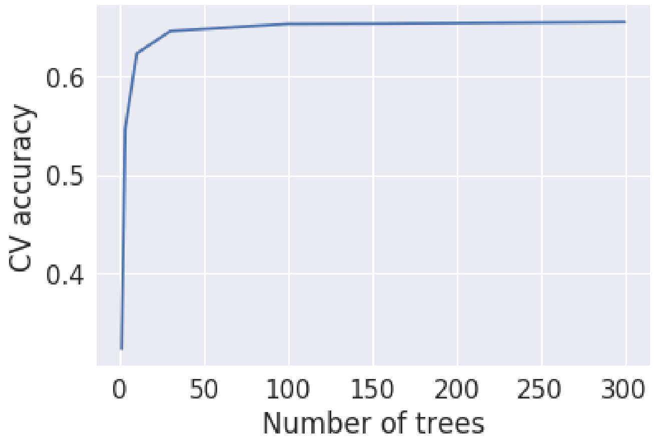
Parameter selection through 3-fold cross-validation: R2

Only the number of trees used in the ensemble (`n_estimators`) is roughly scanned for the initial attempt.

In [14]:

```
param_grid = {'n_estimators':[1,3,10,30,100,150,300]}
grid_rf = GridSearchCV(RandomForestRegressor(),param_
grid,cv=3).fit(X_train,y_train)
plt.figure()
plt.plot(list(param_grid.values())[0],grid_rf.cv_resu
lts_['mean_test_score'])
plt.xlabel('Number of trees')
plt.ylabel('CV accuracy')
print('Best parameter: {}'.format(grid_rf.best_params
_))
print('Best score: {:.2f}'.format(grid_rf.best_score_
))
```

```
Best parameter: {'n_estimators': 300}
Best score: 0.66
```



As the number of tree in RF increase, the average CV R2 score increases as well. While it seems to be saturated at `n_estimators` =~100, it is difficult to tell why the model is suffering with low accuracy score of ~0.65.

By considering `max_depth`, more optimized parameters can be searched up.

In [15]:

```
param_grid = {'n_estimators':[1,3,10,30,100,150,300],
'max_depth':[1,3,5,7,9]}
grid_rf = GridSearchCV(RandomForestRegressor(),param_
grid,cv=3).fit(X_train,y_train)

print('Best parameter: {}'.format(grid_rf.best_params
_))
print('Best score: {:.2f}'.format(grid_rf.best_score_
))
```

```
Best parameter: {'max_depth': 9, 'n_estim
ators': 300}
Best score: 0.66
```

In this specific random grid Search, `max_depth` = 9 and `n_estimators` = 150 were found to be optimal. With the two parameters optimized, however, the 3-fold CV score is still quite low. Next, the learning curve can be considered to understand the performance of the model better.

Model investigation by the learning curve

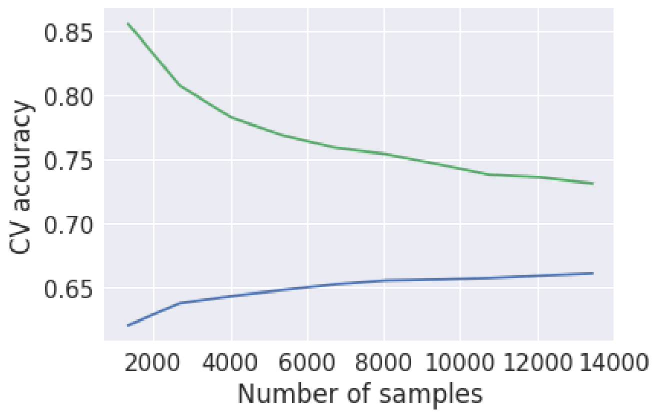
In [16]:

```
train_sizes, train_scores, valid_scores = learning_cu
```

```
rve(RandomForestRegressor(max_depth=9, n_estimators=1
50), X_train, y_train, cv=3, train_sizes =np.linspace
(0.1, 1.0, 10))
plt.figure()
plt.plot(train_sizes,np.mean(valid_scores,axis=1))
plt.plot(train_sizes,np.mean(train_scores,axis=1))
plt.xlabel('Number of samples')
plt.ylabel('CV accuracy')
```

Out[16]:

```
Text(0,0.5,'CV accuracy')
```



From the learning curve, it is shown that the RF model with `max_depth` = 9 and `n_estimators` = 150 is suffering with high variance problem. We may consider investigating the feature importance to remove some features.

Feature importance

In [17]:

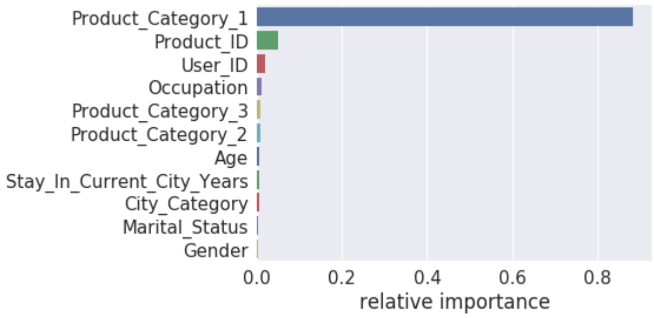
```
rf = RandomForestRegressor(max_depth=9, n_estimators=
150).fit(X_train,y_train)
f_im = rf.feature_importances_.round(3)
ser_rank = pd.Series(f_im,index=X.columns).sort_value
s(ascending=False)
cv_score = cross_val_score(RandomForestRegressor(n_es
timators=45),X_train,y_train,cv=3)
print('CV score: {:.3f}'.format(np.mean(cv_score)))
print('')
print("Feature rank among 7 features:")
print(ser_rank)
plt.figure()
sns.barplot(y=ser_rank.index,x=ser_rank.values,palett
e='deep')
plt.xlabel('relative importance')
```

```
CV score: 0.648

Feature rank among 7 features:
Product_Category_1            0.884
Product_ID                    0.050
User_ID                       0.019
Occupation                    0.010
Product_Category_3            0.009
Product_Category_2            0.009
Age                           0.006
Stay_In_Current_City_Years    0.005
City_Category                 0.005
Marital_Status                0.002
Gender                        0.002
dtype: float64
```

Out[17]:

```
Text(0.5,0,'relative importance')
```



This is somewhat counterintuitive from the results observed from EDA, but we may remove the five least important features (***Gender***, ***Marital_Status***, ***Stay_In_Current_City_Years***, ***Age*** and ***City_Category***) in an attempt to resolve the overfitting issue.

Removing features and 3-fold cross-validation: R2

In [18]:

```
df_frac = df.sample(frac=0.05)
col_model = ['User_ID', 'Product_ID', 'Occupation',
'Product_Category_1', 'Product_Category_2', 'Product_
Category_3']
X = df_frac[col_model]
y = df_frac['Purchase']
X_train,X_test,y_train,y_test = train_test_split(X,y,
random_state=0)
cv_score = cross_val_score(RandomForestRegressor(max_
depth=9, n_estimators=150),X_train,y_train,cv=3)
print('CV score: {:.3f}'.format(np.mean(cv_score)))
train_sizes, train_scores, valid_scores = learning_cu
rve(RandomForestRegressor(max_depth=9, n_estimators=1
```

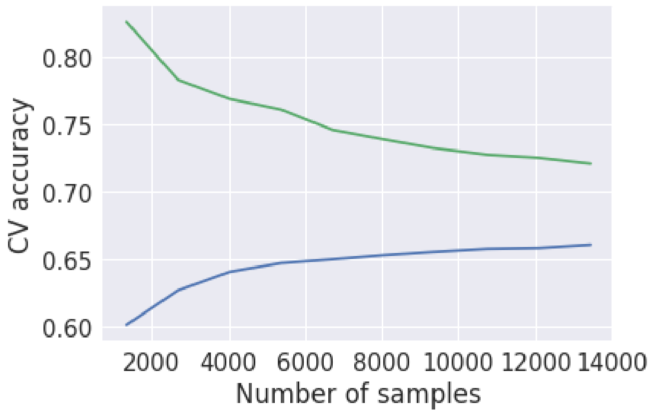
```
rve(RandomForestRegressor(max_depth=9, n_estimators=1
50), X_train, y_train, cv=3,train_sizes =np.linspace(
0.1, 1.0, 10))

plt.figure()
plt.plot(train_sizes,np.mean(valid_scores,axis=1))
plt.plot(train_sizes,np.mean(train_scores,axis=1))
plt.xlabel('Number of samples')
plt.ylabel('CV accuracy')
```

```
CV score: 0.660
```

Out[18]:

```
Text(0,0.5,'CV accuracy')
```



With the five least important features removed, the CV score has marginally improved. Other than removing the feature(s), another way to deal with the high variance problem is to increase the number of training set. Fortunately, we have tons of data available for training. Even though it will take much longer to fit the model, it is worthy to try.

Utilize the entire dataset for 3-fold cross-validation: R2
(the test set is totally isolated)

In [19]:

```
df_frac = df.sample(frac=1)
col_model = ['User_ID', 'Product_ID', 'Age', 'Occupat
ion', 'Stay_In_Current_City_Years', 'Product_Category
_1', 'Product_Category_2', 'Product_Category_3']
X = df_frac[col_model]
y = df_frac['Purchase']
X_train,X_test,y_train,y_test = train_test_split(X,y,
random_state=0)
```

Did you find this Kernel useful?
Show your appreciation with an upvote

1

Comments (0)

Sort by Select...

Click here to enter a comment...