Limitations

Before attempting the problem, I want to list down the limitations of this problem and how they affect the analysis:

1. The fields in the data do not have any description. All the naked eye can see is the presence of 7 continuous variables and 2 categorical variables. With only this information about the data and no well-defined problem statement, it is not possible to discard any variable using the doamin knowledge. Only some statistical feature selection methods can be implemented at later stage to optimize our model.
2. There are only 500 observations in our training data, and that might mean that our model will not have enough data to 'learn'.
3. Also, since there is no data description given, I have built by model with accuracy as my sole focus.

Data Exploration

In [74]:
```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn import metrics
from sklearn.feature_selection import RFE
from sklearn.linear_model import RidgeCV, LassoCV, Ridge, Lasso
import seaborn as sns
import statsmodels.api as sm
```

In [2]:
```
df_intern = pd.read_csv('intern_data.csv')
```

In [3]:
```
df_intern.head()
```

Out[3]:

	Unnamed: 0	a	b	c	d	e	f	g	h	
0	7	0.951786	0.669570	green	0.170130	0.623469	0.925886	0.812685	white	3.70751
1	43	0.510447	0.922627	red	0.087899	0.025415	0.698444	0.658545	white	2.68924
2	47	0.294838	0.351081	yellow	0.710892	0.699661	0.545722	0.836863	black	2.88650
3	53	0.798645	0.572042	green	0.026137	0.609730	0.488668	0.342675	black	2.47816
4	54	0.689666	0.395323	red	0.172448	0.736433	0.708408	0.695521	white	3.18266

The question asks us to build a model using on a-h variables, so we can get rid of the unnamed column at the start.

```
In [8]: del df_intern['Unnamed: 0']
```

Apart from the removal of this column, everything else in the data looks good to me. Naturally, not all of these features will be used for actual regression, and thus we will have to perform feature selection to obtain the best predictors. However, apart from this, everything else about the data looks clean, so I am assuming that the goal for this exercise is not to perform data cleaning (which is the task in Data Science process that takes the maximum amount of time).

Let us have a look at the data types for all the fields.

```
In [9]: df_intern.dtypes
```

```
Out[9]: a    float64
        b    float64
        c     object
        d    float64
        e    float64
        f    float64
        g    float64
        h     object
        y    float64
        dtype: object
```

Now let us look at the size of our dataset.

```
In [10]: df_intern.shape
```

```
Out[10]: (500, 9)
```

Finally, let us run some descriptive analysis for our data.

```
In [18]: df_intern.describe()
```

Out[18]:

	a	b	d	e	f	g	y
count	500.000000	500.000000	500.000000	500.000000	500.000000	500.000000	500.000000
mean	0.497413	0.505397	0.501722	0.501899	0.526562	0.508442	2.682803
std	0.292947	0.292450	0.284623	0.272019	0.292175	0.284332	0.592533
min	0.002801	0.000369	0.000592	0.002387	0.001327	0.005644	0.981376
25%	0.253897	0.256446	0.252244	0.270069	0.262523	0.263525	2.300795
50%	0.510420	0.514069	0.496800	0.515185	0.506412	0.526403	2.726900
75%	0.760192	0.766578	0.735650	0.732240	0.790898	0.742125	3.155562
max	0.999690	0.997025	0.998282	0.998994	0.999271	0.998156	3.980509

```
In [20]: print(df_intern['c'].value_counts())
         print(df_intern['h'].value_counts())
```

```
yellow    134
green     124
blue      123
red       119
Name: c, dtype: int64
white    392
black    108
Name: h, dtype: int64
```

It is clear that the distribution of the 'c' variable classes is pretty event. However, in case of the 'h' variable, the counts are highly biased towards the white.

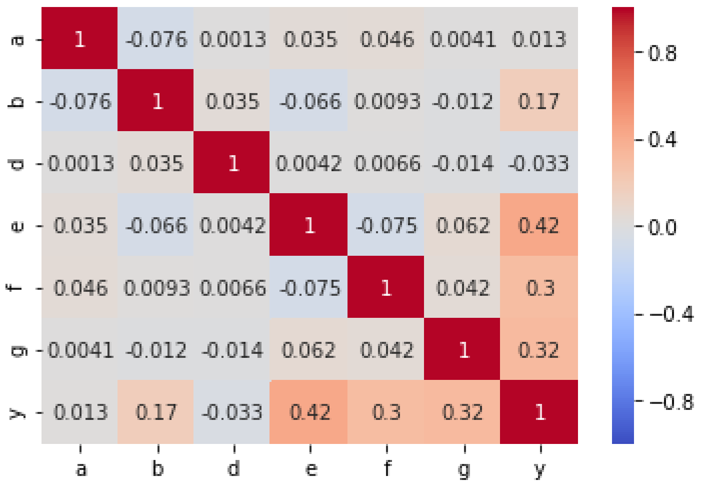
Visualization of the correlation between the variables and their distributions

Before we start building a linear regression model, there are several key assumptions that we need to take care of:

1. Variables are normally distributed: Non-normally distributed variables (highly skewed variables or variables with substantial outliers) can distort relationships and significance tests.
2. Linear relationship between the predictor and the response: Standard multiple regression can only accurately estimate the relationship between dependent and independent variables if the relationships are linear in nature. If the relationship between each of the predictors and the response is not linear, the results of the regression analysis will under-estimate the true relationship.
3. Variables are measured without error: Effect sizes of other variables can be over-estimated if the covariate is not reliably measured, as the full effect of the covariate(s) would not be removed.
4. Absence of homoscedasticity: Homoscedasticity means that the variance of errors is the same across all levels of the IV. It can lead to serious distortion of findings and seriously weaken the analysis.

In [27]:
```
#Correlation heatmap
sns.heatmap(df_intern.corr(), annot = True, vmin=-1, vmax=1, center= 0, cmap=
'coolwarm')
```

Out[27]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x280439f0080>
```

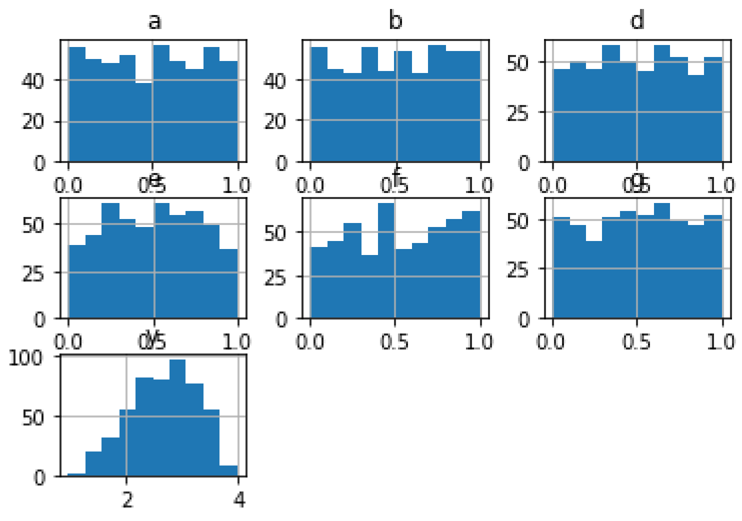


The scatter plot above shows that there is a decent amount of correlation between the predictors (a-h) and the response variable (y), with the clear exception of variable a and d with y.

One good thing is that there is no linear relationship of the predictors with each other. Presence of such a linear relationship will give incorrect results during regression.

```
In [42]: df_intern.hist()
```

```
Out[42]: array([[<matplotlib.axes._subplots.AxesSubplot object at 0x000002804B1AD9E8>,
                 <matplotlib.axes._subplots.AxesSubplot object at 0x000002804BC7C2E8>,
                 <matplotlib.axes._subplots.AxesSubplot object at 0x000002804BCA4898
         >],
                [<matplotlib.axes._subplots.AxesSubplot object at 0x000002804BBA4E48>,
                 <matplotlib.axes._subplots.AxesSubplot object at 0x000002804B9E3438>,
                 <matplotlib.axes._subplots.AxesSubplot object at 0x000002804B62D860
         >],
                [<matplotlib.axes._subplots.AxesSubplot object at 0x000002804BB46F28>,
                 <matplotlib.axes._subplots.AxesSubplot object at 0x000002804BAA60F0>,
                 <matplotlib.axes._subplots.AxesSubplot object at 0x000002804BAA6128
         >]],
               dtype=object)
```

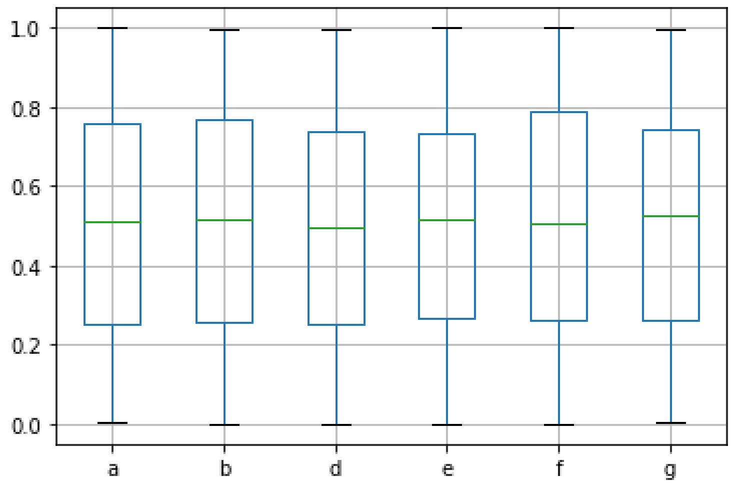


The variables should be noramally distributed. We see that our vairables a-h are not really normally distributed, so this assumption of linear regression is disobeyed.

```
In [41]: #Boxplots

         df_intern.boxplot(column=list('abdefg'))
```

```
Out[41]: <matplotlib.axes._subplots.AxesSubplot at 0x2804be50f28>
```



The boxplots show that the means of all the distributions are fairly similar. In fact, even the 1st and 3rd quartiles of the distributions are very close to each other.

Feature Selection - Choosing the best predictors

Why do we even need feature selection?

1. It makes the machine learning model faster.
2. It improves the accuracy of a model.
3. It reduces the complexity of a model and makes it more interpretable.
4. It reduces Overfitting in our model.

There are three broad types of feature selection methods that I know:

1. Filter based methods
2. Wrapper methods
3. Embedded methods

The feature selection methods we will use here all come under what is called a 'Wrapper Method'. Wrapper feature selection methods create many models with different subsets of input features and select those features that result in the best performing model according to a performance metric. This is an iterative and computationally expensive process but since we have very less number of features, we can use this method.

The wrapper methods we will use here are Backward selection and RFE.

```
In [71]: X = pd.get_dummies(df_intern).drop(['y'], axis=1)
         y = df_intern['y']
```

In backward feature selection method, we feed all the possible features to the model, check the performance of the model and then iteratively remove the worst performing features one by one till the overall performance of the model comes in acceptable range. The performance metric used here to evaluate feature performance is pvalue.

In [73]:
```
cols = list(X.columns)
pmax = 1
while (len(cols)>0):
    p= []
    X_1 = X[cols]
    X_1 = sm.add_constant(X_1)
    model = sm.OLS(y,X_1).fit()
    p = pd.Series(model.pvalues.values[1:],index = cols)
    pmax = max(p)
    feature_with_p_max = p.idxmax()
    if(pmax>0.05):
        cols.remove(feature_with_p_max)
    else:
        break

selected_features_BE = cols
print(selected_features_BE)
```

```
['b', 'e', 'f', 'g', 'c_blue', 'c_green', 'c_red', 'c_yellow', 'h_white']
```

The Recursive Feature Elimination (RFE) method works by recursively removing attributes and building a model on those attributes that remain. It uses accuracy metric to rank the feature according to their importance.

Instead of taking the number of features randomly, we can find the optimum number of features by using loop starting with 1 feature and going up to 12 features and then choose the one for which the accuracy is highest.

In [83]:
```
#no of features
nof_list=np.arange(1,12)
high_score=0
#Variable to store the optimum features
nof=0
score_list =[]
for n in range(len(nof_list)):
    X_train, X_test, y_train, y_test = train_test_split(X,y, test_size = 0.3,
random_state = 0)
    model = LinearRegression()
    rfe = RFE(model,nof_list[n])
    X_train_rfe = rfe.fit_transform(X_train,y_train)
    X_test_rfe = rfe.transform(X_test)
    model.fit(X_train_rfe,y_train)
    score = model.score(X_test_rfe,y_test)
    score_list.append(score)
    if(score>high_score):
        high_score = score
        nof = nof_list[n]

print("Optimum number of features: %d" %nof)
print("Score with %d features: %f" % (nof, high_score))
```

```
Optimum number of features: 9
Score with 9 features: 0.972735
```

Now that we have an optimum number of features, we can implement RFE to obtain these 9 best features.

In [85]:
```
cols = list(X.columns)
model = LinearRegression()

#Initializing RFE model
rfe = RFE(model, 9)

#Transforming data using RFE
X_rfe = rfe.fit_transform(X,y)

#Fitting the data to model
model.fit(X_rfe,y)
temp = pd.Series(rfe.support_,index = cols)
selected_features_rfe = temp[temp==True].index

print(selected_features_rfe)
```

```
Index(['b', 'e', 'f', 'g', 'c_blue', 'c_green', 'c_yellow', 'h_black',
       'h_white'],
      dtype='object')
```

Cross-Validation

Cross-validation is the best way to evaluate models used for prediction. Here you divide your data set into two group (train and validate). A simple mean squared difference between the observed and predicted values give you a measure for the prediction accuracy.

```
In [ ]:
```

Building Regression Models

By now we are already clear that since we have a label in our data, it is a supervised learning problem. Also, since we are predicting a continuous variable, it is a regression problem.

So with this information, we now have to decide which algorithm to finally go with. Now idedally, there are several things I like to check before selecting any model:

1. Checking if the model aligns to the business goals
2. The amount of pre-processing before building the model
3. How interpretable the model is
4. How long does it take to build a model, and how long does the model take to make predictions
5. Checking the accuracy of the model
6. The scalability of the model

Now for our current problem, we do not have any business goals defined, so we shall skip the first check. Next, we see that the data given to us was already clean, so the data cleaning section of pre-processing isn't needed either. The only pre-processing that we did was feature reduction. This ensures that the pre-processing for our current data will not take a lot of time. Without any data description, the interpretability of the model also becomes difficult to test. Easiest and most basic way to explain our model will be that we are trying to fit a parametric function on our data to predict a continous response variable.

Regarding the speed of the model, we first need to check the model complexity. The model will use only 9 features, and the amount of training data will also not be over 500 (even if we decide to use all our available data for training). Additionally, we also haven't used any complex feature engineering methods like principal component analysis on our data for dimensionality reduction. Accuracy is something we will be testing in the code below. The scalability of the model is something that is out of scope for our current problem.

For the sake of simplicity, I will only compare the following regression based algorithms:

1. Multiple Linear Regression
2.

Since there is not much multicolinearity or high dimensionality in our data, we do not unnecessarily use any regression regularization methods like Ridge, Lasso or ElasticNet. Similarly, since we have only one response variable, multivariate regression is also out of picture.

Multiple Linear Regression

```
In [108]: X_train, X_test, y_train, y_test = train_test_split(X[selected_features_rfe],
          y, test_size=0.2, random_state=0)

          lm = LinearRegression()
          lm.fit(X_train, y_train)

          print(lm.intercept_)
          print(lm.coef_)

          y_pred = lm.predict(X_test)

          np.sqrt(metrics.mean_squared_error(y_test, y_pred))
```

```
1.1405634450995108
[ 0.39188035  0.96160952  0.66608727  0.60155309 -0.56813786  0.32658911
  0.31747677 -0.31818715  0.31818715]
```

```
Out[108]: 0.10004774125046247
```

```
In [109]: #Rsquared value

          lm.score(X_train, y_train)
```

```
Out[109]: 0.9726124630602735
```

Random Forest for Regression

```
In [95]: # Fitting Random Forest Regression to the dataset
         # import the regressor
         from sklearn.ensemble import RandomForestRegressor

          # create regressor object
         regressor = RandomForestRegressor(n_estimators = 100, random_state = 42)

         # fit the regressor with x and y data
         regressor.fit(X_train, y_train)

         y_pred = regressor.predict(X_test)

         print(np.sqrt(metrics.mean_squared_error(y_test, y_pred)))
```

```
0.22935549491883936
```

In [105]:
```
#Variable Importance

importances = list(regressor.feature_importances_)

feature_importances = [(feature, round(importance, 2)) for feature, importance
in zip(X_test, importances)]
feature_importances = sorted(feature_importances, key = lambda x: x[1], revers
e = True)

print([{a:b} for a,b in feature_importances])
```

```
[{'c_blue': 0.26}, {'e': 0.24}, {'f': 0.12}, {'h_white': 0.12}, {'g': 0.11},
{'h_black': 0.1}, {'b': 0.04}, {'c_green': 0.01}, {'c_yellow': 0.01}]
```

In [106]:
```
# New random forest with only the two most important variables
rf_most_important = RandomForestRegressor(n_estimators= 1000, random_state=42)

train_important = X_train[['c_blue','e']]
test_important = X_test[['c_blue','e']]

rf_most_important.fit(train_important, y_train)
predictions = rf_most_important.predict(test_important)

print(np.sqrt(metrics.mean_squared_error(y_test, predictions)))
```

```
0.5743071742446706
```

References

Throughout this analysis, I have used several sources for revising concepts. I have been bookmarking these sources over the last few years, and I take opportunities like these to revise them. I've added them below so that they get the credit they deserve:

1. Feature Selection https://towardsdatascience.com/feature-selection-with-pandas-e3690ad8504b (https://towardsdatascience.com/feature-selection-with-pandas-e3690ad8504b)
2. Assumptions of Linear Regression https://scholarworks.umass.edu/cgi/viewcontent.cgi?article=1111&context=pare (https://scholarworks.umass.edu/cgi/viewcontent.cgi?article=1111&context=pare)

In [ ]: