In [1]:
```
# I will use in this Kernel the step-by-step process of Will Koehrsen.
# I won't use everything, but most of them.
# This project at in GitHub repository: https://github.com/WillKoehrsen/machine
```

In [2]:
```
# Let's import the main libraries that I will use in this dataset.

# Pandas and numpy for data manipulation
import pandas as pd
import numpy as np

# No warnings about setting value on copy of slice
pd.options.mode.chained_assignment = None

# Display up to 9 columns of a dataframe
pd.set_option('display.max_columns', 9)

# Matplotlib visualization
import matplotlib.pyplot as plt
%matplotlib inline

# Set default font size
plt.rcParams['font.size'] = 24

# Internal ipython tool for setting figure size
from IPython.core.pylabtools import figsize

# Seaborn for visualization
import seaborn as sns
sns.set(font_scale = 2)

# Splitting data into training and testing
from sklearn.model_selection import train_test_split
```

In [38]:
```
# I will check the two CSV files to see what the difference between them.

graduate_first = pd.read_csv('Admission_Predict.csv')
graduate_first.head()
```

Out[38]:

	Serial No.	GRE Score	TOEFL Score	University Rating	SOP	LOR	CGPA	Research	Chance of Admit
0	1	337	118	4	4.5	4.5	9.65	1	0.92
1	2	324	107	4	4.0	4.5	8.87	1	0.76
2	3	316	104	3	3.0	3.5	8.00	1	0.72
3	4	322	110	3	3.5	2.5	8.67	1	0.80
4	5	314	103	2	2.0	3.0	8.21	0	0.65

In [39]:
```
graduate_first.shape
```

Out[39]:
```
(400, 9)
```

In [40]:
```
graduate_second = pd.read_csv('Admission_Predict.csv')
graduate_second.head()
```

Out[40]:

	Serial No.	GRE Score	TOEFL Score	University Rating	SOP	LOR	CGPA	Research	Chance of Admit
0	1	337	118	4	4.5	4.5	9.65	1	0.92
1	2	324	107	4	4.0	4.5	8.87	1	0.76
2	3	316	104	3	3.0	3.5	8.00	1	0.72
3	4	322	110	3	3.5	2.5	8.67	1	0.80
4	5	314	103	2	2.0	3.0	8.21	0	0.65

In [41]:
```
graduate_second.shape
```

Out[41]:
```
(400, 9)
```

In [42]:
```
# I saw that maybe the second one is a recent version, so I will use this one.
graduate = pd.read_csv('Admission_Predict.csv')

# I will drop the 'Serial No' because it's not important for our model.
graduate.drop(labels='Serial No.', axis=1, inplace=True)
```

In [12]:
```
# See the column data types and non-missing values.
graduate.info()

# Apparently we don't have any missing values;
# We don't have any 'object' column to convert to 'float' or 'int'.
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 400 entries, 0 to 399
Data columns (total 8 columns):
 #   Column             Non-Null Count  Dtype  
---  ------             --------------  -----  
 0   GRE Score          400 non-null    int64  
 1   TOEFL Score        400 non-null    int64  
 2   University Rating  400 non-null    int64  
 3   SOP                400 non-null    float64
 4   LOR                400 non-null    float64
 5   CGPA               400 non-null    float64
 6   Research           400 non-null    int64  
 7   Chance of Admit    400 non-null    float64
dtypes: float64(4), int64(4)
memory usage: 25.1 KB
```

In [13]:
```
# Statistics for each column

graduate.describe()
```

Out[13]:

	GRE Score	TOEFL Score	University Rating	SOP	LOR	CGPA	Research	Cha
count	400.000000	400.000000	400.000000	400.000000	400.000000	400.000000	400.000000	400.0
mean	316.807500	107.410000	3.087500	3.400000	3.452500	8.598925	0.547500	0.7
std	11.473646	6.069514	1.143728	1.006869	0.898478	0.596317	0.498362	0.1
min	290.000000	92.000000	1.000000	1.000000	1.000000	6.800000	0.000000	0.3
25%	308.000000	103.000000	2.000000	2.500000	3.000000	8.170000	0.000000	0.6
50%	317.000000	107.000000	3.000000	3.500000	3.500000	8.610000	1.000000	0.7
75%	325.000000	112.000000	4.000000	4.000000	4.000000	9.062500	1.000000	0.8
max	340.000000	120.000000	5.000000	5.000000	5.000000	9.920000	1.000000	0.9

In [14]:
```
# with the pourpuse to be sure about no missing values in our dataset. I will c

# Function to calculate missing values by column
def missing_values_table(df):
        # Total missing values
        mis_val = df.isnull().sum()

        # Percentage of missing values
        mis_val_percent = 100 * df.isnull().sum() / len(df)

        # Make a table with the results
        mis_val_table = pd.concat([mis_val, mis_val_percent], axis=1)

        # Rename the columns
        mis_val_table_ren_columns = mis_val_table.rename(
        columns = {0 : 'Missing Values', 1 : '% of Total Values'})

        # Sort the table by percentage of missing descending
        mis_val_table_ren_columns = mis_val_table_ren_columns[
            mis_val_table_ren_columns.iloc[:,1] != 0].sort_values(
        '% of Total Values', ascending=False).round(1)

        # Print some summary information
        print ("Your selected dataframe has " + str(df.shape[1]) + " columns.\r
            "There are " + str(mis_val_table_ren_columns.shape[0]) +
              " columns that have missing values.")

        # Return the dataframe with missing information
        return mis_val_table_ren_columns
```

In [15]:
```
missing_values_table(graduate)

# Great! Now we now that for sure we don't have any missing values.
```

```
Your selected dataframe has 8 columns.
There are 0 columns that have missing values.
```

Out[15]:

	Missing Values	% of Total Values

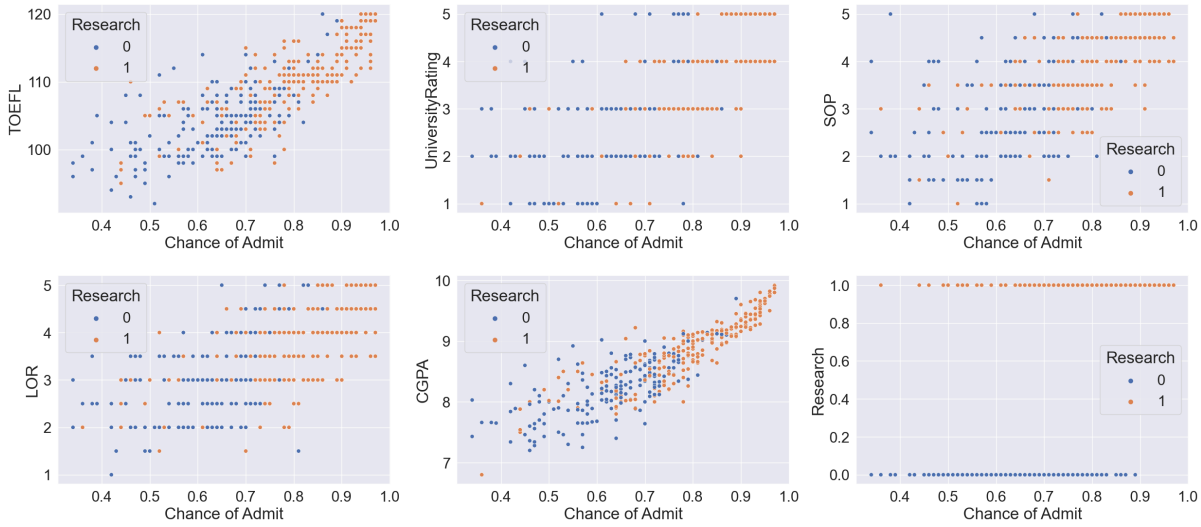
In [16]:
```
# Let's start now the Exploratory Data Analysis (EDA) to understand better our

# Fist I will see the name of the columns. The goal here is read the name of th
# Sometimes is a good practice to rename some of them to easy manipulate.

graduate.columns

# We can see below that some column names has a space in the end, is good to re
# To manipule better the columns, I will chance the name of some of them as wee
```

Out[16]:
```
Index(['GRE Score', 'TOEFL Score', 'University Rating', 'SOP', 'LOR ', 'CGP
A',
       'Research', 'Chance of Admit '],
      dtype='object')
```

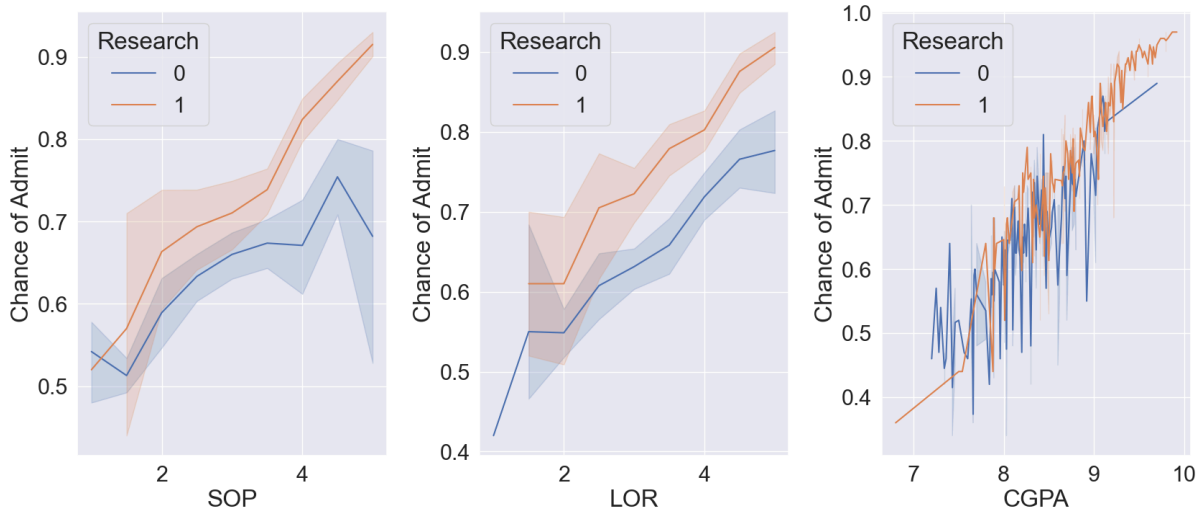
In [17]:
```
graduate.rename(columns = {'Serial No.': 'SerialNo', 'GRE Score': 'GRE', 'TOEFL
graduate.columns
```

Out[17]:
```
Index(['GRE', 'TOEFL', 'UniversityRating', 'SOP', 'LOR', 'CGPA', 'Research',
       'Chance'],
      dtype='object')
```

In [18]:
```
# First of all, I will see the correlation between any variable with the target

# I will drop the 'SerialNo' and 'Research' columns because the serial number j
# use in the 'hue' parameter.

fig = plt.figure(figsize=(30,20))
fig.subplots_adjust(hspace=0.3, wspace=0.2)
for i in range(1, 7):
    ax = fig.add_subplot(3, 3, i)
    sns.scatterplot(x=graduate['Chance'], y= graduate.iloc[:,i], hue=graduate.R
    plt.xlabel('Chance of Admit')
    plt.ylabel(graduate.columns[i])

# Conclusions:
#     - The better graph of the features 'UniversityRating', 'SOP', 'LOR' and 'R
#     - 'GRE', 'TOEFL' and 'CGPA' graphs have a linear behavior;
#     - The tendency which we can see is, as higher as the 'GRE', 'TOEFL' and 'C
#     - The other tendency that we can see is if the person has a research has m
```



In [19]:
```
fig = plt.figure(figsize=(20,8))
fig.subplots_adjust(hspace=0.1, wspace=0.3)
for i in range(1, 4):
    ax = fig.add_subplot(1, 3, i)
    sns.lineplot(x= graduate.iloc[:,i+2], y= graduate['Chance'], hue=graduate.R
    plt.xlabel(graduate.columns[i+2])
    plt.ylabel('Chance of Admit')

# Conclusion:
#     - Here we can see again a linear correlation between these variables and t
#     - The tendency which we can see is, as higher as the 'UniversityRating', '
#     - The other tendency that we can see is if the person has a research has m
```



In [22]:
```
# Now we will remove the outliers

# I will use a stats concept (formula) to figure out the outliers that maybe ca

for i in graduate.columns:
    # Calculate first and third quartile
    first_quartile = graduate[i].describe()['25%']
    third_quartile = graduate[i].describe()['75%']

    # Interquartile range
    iqr = third_quartile - first_quartile

    # Remove outliers
    graduate = graduate[(graduate[i] > (first_quartile - 3 * iqr)) & (graduate[
```

In [23]:
```
# Let's quantify the correlations between the features with the target and see

# Find all correlations and sort
correlations_data = graduate.corr()['Chance'].sort_values(ascending=False)

# Print the correlations
print(correlations_data)

# Conclusions:
#    - We have basic three groups of influencers: high(CGPA, GRE and TOEFL), in
#    - All of them have a positive influence.
```

```
Chance              1.000000
CGPA                0.873289
GRE                 0.802610
TOEFL               0.791594
UniversityRating    0.711250
SOP                 0.675732
LOR                 0.669889
Research            0.553202
Name: Chance, dtype: float64
```

In [24]:
```
# # # Split Into Training and Testing Sets

# Separate out the features and targets
features = graduate.drop(columns='Chance')
targets = pd.DataFrame(graduate['Chance'])

# Split into 70% training and 30% testing set
X_train, X_test, y_train, y_test = train_test_split(features, targets, test_siz

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

```
(320, 7)
(80, 7)
(320, 1)
(80, 1)
```

In [25]:
```
# # # Establish a Baseline

# # Metric: Mean Absolute Error

# Function to calculate mean absolute error
def mae(y_true, y_pred):
    return np.mean(abs(y_true - y_pred))
```

In [26]:
```
# Now we can make the median guess and evaluate it on the test set.
baseline_guess = np.median(y_train)

print('The baseline guess is a score of %0.2f' % baseline_guess)
print("Baseline Performance on the test set: MAE = %0.4f" % mae(y_test, baselir
```

```
The baseline guess is a score of 0.73
Baseline Performance on the test set: MAE = 0.1314
```

In [27]:
```
# # # Feature Scaling

from sklearn.preprocessing import StandardScaler
sc_X = StandardScaler()
X_train = sc_X.fit_transform(X_train)
X_test = sc_X.transform(X_test)
```

In [28]:
```
# # # Evaluating and Comparing Machine Learning Models

# Imputing missing values and scaling values
from sklearn.preprocessing import Imputer, MinMaxScaler

# Machine Learning Models
from sklearn.linear_model import LinearRegression
from sklearn.ensemble import RandomForestRegressor, GradientBoostingRegressor
from sklearn.svm import SVR
from sklearn.neighbors import KNeighborsRegressor
```

```
---------------------------------------------------------------------------
ImportError                               Traceback (most recent call last)
Cell In[28], line 4
      1 # # # Evaluating and Comparing Machine Learning Models
      2
      3 # Imputing missing values and scaling values
----> 4 from sklearn.preprocessing import Imputer, MinMaxScaler
      6 # Machine Learning Models
      7 from sklearn.linear_model import LinearRegression

ImportError: cannot import name 'Imputer' from 'sklearn.preprocessing' (D:\an
aconda\Lib\site-packages\sklearn\preprocessing\__init__.py)
```

In [29]:
```
# Create an imputer object with a median filling strategy
imputer = Imputer(strategy='median')

# Train on the training features
imputer.fit(X_train)

# Transform both training data and testing data
X_train = imputer.transform(X_train)
X_test = imputer.transform(X_test)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[29], line 2
      1 # Create an imputer object with a median filling strategy
----> 2 imputer = Imputer(strategy='median')
      4 # Train on the training features
      5 imputer.fit(X_train)

NameError: name 'Imputer' is not defined
```

In [30]:
```
# Convert y to one-dimensional array (vector)
y_train = np.array(y_train).reshape((-1, ))
y_test = np.array(y_test).reshape((-1, ))
```

In [31]:
```
# # # Models to Evaluate

# We will compare five different machine learning models:

# 1 - Linear Regression
# 2 - Support Vector Machine Regression
# 3 - Random Forest Regression
# 4 - Gradient Boosting Regression
# 5 - K-Nearest Neighbors Regression

# Function to calculate mean absolute error
def mae(y_true, y_pred):
    return np.mean(abs(y_true - y_pred))

# Takes in a model, trains the model, and evaluates the model on the test set
def fit_and_evaluate(model):

    # Train the model
    model.fit(X_train, y_train)

    # Make predictions and evalute
    model_pred = model.predict(X_test)
    model_mae = mae(y_test, model_pred)

    # Return the performance metric
    return model_mae
```

In [32]:
```
# # Linear Regression

lr = LinearRegression()
lr_mae = fit_and_evaluate(lr)

print('Linear Regression Performance on the test set: MAE = %0.4f' % lr_mae)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[32], line 3
      1 # # Linear Regression
----> 3 lr = LinearRegression()
      4 lr_mae = fit_and_evaluate(lr)
      6 print('Linear Regression Performance on the test set: MAE = %0.4f' %
lr_mae)

NameError: name 'LinearRegression' is not defined
```

In [33]:
```
# # SVM

svm = SVR(C = 1000, gamma = 0.1)
svm_mae = fit_and_evaluate(svm)

print('Support Vector Machine Regression Performance on the test set: MAE = %0.
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[33], line 3
      1 # # SVM
----> 3 svm = SVR(C = 1000, gamma = 0.1)
      4 svm_mae = fit_and_evaluate(svm)
      6 print('Support Vector Machine Regression Performance on the test set:
MAE = %0.4f' % svm_mae)

NameError: name 'SVR' is not defined
```

In [28]:
```
# # Random Forest

random_forest = RandomForestRegressor(random_state=60)
random_forest_mae = fit_and_evaluate(random_forest)

print('Random Forest Regression Performance on the test set: MAE = %0.4f' % rar
```

```
Random Forest Regression Performance on the test set: MAE = 0.0492

/opt/conda/lib/python3.6/site-packages/sklearn/ensemble/forest.py:246: Future
Warning: The default value of n_estimators will change from 10 in version 0.2
0 to 100 in 0.22.
  "10 in version 0.20 to 100 in 0.22.", FutureWarning)
```

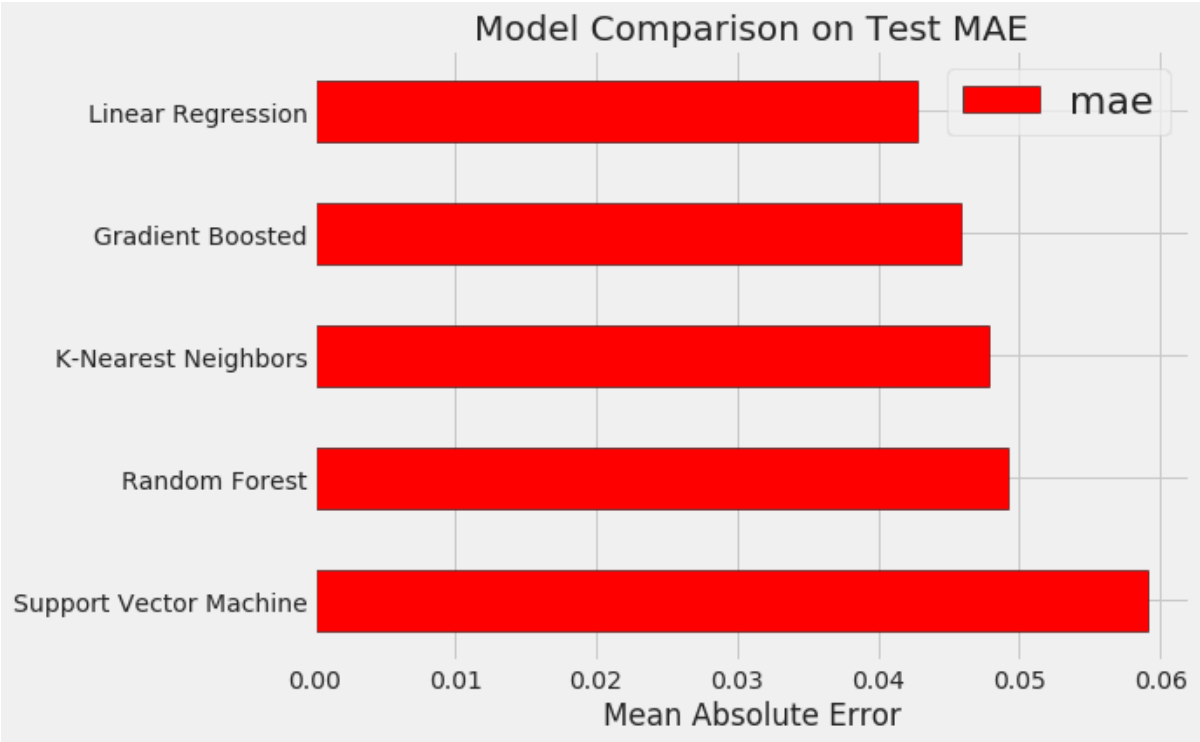
In [29]:
```
# # Gradiente Boosting Regression

gradient_boosted = GradientBoostingRegressor(random_state=60)
gradient_boosted_mae = fit_and_evaluate(gradient_boosted)

print('Gradient Boosted Regression Performance on the test set: MAE = %0.4f' %
```

```
Gradient Boosted Regression Performance on the test set: MAE = 0.0459
```

In [21]:
```
# # KNN

knn = KNeighborsRegressor(n_neighbors=10)
knn_mae = fit_and_evaluate(knn)

print('K-Nearest Neighbors Regression Performance on the test set: MAE = %0.4f'
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[21], line 3
      1 # # KNN
----> 3 knn = KNeighborsRegressor(n_neighbors=10)
      4 knn_mae = fit_and_evaluate(knn)
      6 print('K-Nearest Neighbors Regression Performance on the test set: MA
E = %0.4f' % knn_mae)

NameError: name 'KNeighborsRegressor' is not defined
```

In [31]:
```
# Now, to better understand the results, I will show in a graph the model that

plt.style.use('fivethirtyeight')
figsize(8, 6)

# Dataframe to hold the results
model_comparison = pd.DataFrame({'model': ['Linear Regression', 'Support Vector
                                           'Random Forest', 'Gradient Boosted',
                                            'K-Nearest Neighbors'],
                                 'mae': [lr_mae, svm_mae, random_forest_mae,
                                         gradient_boosted_mae, knn_mae]})

# Horizontal bar chart of test mae
model_comparison.sort_values('mae', ascending = False).plot(x = 'model', y = 'm
                                                            color = 'red', edgec

# Plot formatting
plt.ylabel(''); plt.yticks(size = 14); plt.xlabel('Mean Absolute Error'); plt.x
plt.title('Model Comparison on Test MAE', size = 20);
```



In [32]:
```
# # # Model Optimization

# # Hyperparameter

# Hyperparameter Tuning with Random Search and Cross Validation

# Here we will implement random search with cross validation to select the opti
# We first define a grid then peform an iterative process of: randomly sample a
# and then select the hyperparameters with the best performance.

# Loss function to be optimized
loss = ['ls', 'lad', 'huber']

# Number of trees used in the boosting process
n_estimators = [100, 500, 900, 1100, 1500]

# Maximum depth of each tree
max_depth = [2, 3, 5, 10, 15]

# Minimum number of samples per leaf
min_samples_leaf = [1, 2, 4, 6, 8]

# Minimum number of samples to split a node
min_samples_split = [2, 4, 6, 10]

# Maximum number of features to consider for making splits
max_features = ['auto', 'sqrt', 'log2', None]

# Define the grid of hyperparameters to search
hyperparameter_grid = {'loss': loss,
                       'n_estimators': n_estimators,
                       'max_depth': max_depth,
                       'min_samples_leaf': min_samples_leaf,
                       'min_samples_split': min_samples_split,
                       'max_features': max_features}
```

In [33]:
```
# In the code below, we create the Randomized Search Object passing in the foll

#    estimator: the model
#    param_distributions: the distribution of parameters we defined
#    cv the number of folds to use for k-fold cross validation
#    n_iter: the number of different combinations to try
#    scoring: which metric to use when evaluating candidates
#    n_jobs: number of cores to run in parallel (-1 will use all available)
#    verbose: how much information to display (1 displays a limited amount)
#    return_train_score: return the training score for each cross-validation fo
#    random_state: fixes the random number generator used so we get the same re
```

In [34]:
```
# The Randomized Search Object is trained the same way as any other scikit-lear
# After training, we can compare all the different hyperparameter combinations

# Create the model to use for hyperparameter tuning
model = GradientBoostingRegressor(random_state = 42)

# Set up the random search with 4-fold cross validation
from sklearn.model_selection import RandomizedSearchCV, GridSearchCV
random_cv = RandomizedSearchCV(estimator=model,
                               param_distributions=hyperparameter_grid,
                               cv=4, n_iter=25,
                               scoring = 'neg_mean_absolute_error',
                               n_jobs = -1, verbose = 1,
                               return_train_score = True,
                               random_state=42)
```

In [35]:
```
# Fit on the training data
random_cv.fit(X_train, y_train)
```

```
Fitting 4 folds for each of 25 candidates, totalling 100 fits

[Parallel(n_jobs=-1)]: Using backend LokyBackend with 4 concurrent workers.
[Parallel(n_jobs=-1)]: Done  42 tasks      | elapsed:   28.1s
[Parallel(n_jobs=-1)]: Done 100 out of 100 | elapsed:   53.0s finished
```

Out[35]:
```
RandomizedSearchCV(cv=4, error_score='raise-deprecating',
          estimator=GradientBoostingRegressor(alpha=0.9, criterion='friedman_
mse', init=None,
             learning_rate=0.1, loss='ls', max_depth=3, max_features=None,
             max_leaf_nodes=None, min_impurity_decrease=0.0,
             min_impurity_split=None, min_samples_leaf=1,
             min_sampl...te=42, subsample=1.0, tol=0.0001,
             validation_fraction=0.1, verbose=0, warm_start=False),
          fit_params=None, iid='warn', n_iter=25, n_jobs=-1,
          param_distributions={'loss': ['ls', 'lad', 'huber'], 'n_estimator
s': [100, 500, 900, 1100, 1500], 'max_depth': [2, 3, 5, 10, 15], 'min_samples
_leaf': [1, 2, 4, 6, 8], 'min_samples_split': [2, 4, 6, 10], 'max_features':
['auto', 'sqrt', 'log2', None]},
          pre_dispatch='2*n_jobs', random_state=42, refit=True,
          return_train_score=True, scoring='neg_mean_absolute_error',
          verbose=1)
```

In [36]:
```
# Scikit-learn uses the negative mean absolute error for evaluation because it
# Therefore, a better score will be closer to 0. We can get the results of the

# Get all of the cv results and sort by the test performance
random_results = pd.DataFrame(random_cv.cv_results_).sort_values('mean_test_sc

random_results.head(10)
```

Out[36]:

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	...	split2_train_score	split3_tra
17	0.515520	0.009249	0.001964	0.000062	...	-0.029068	-
24	0.413290	0.015437	0.001249	0.000164	...	-0.019549	-
19	1.373520	0.057453	0.003428	0.000083	...	-0.023699	-
12	1.188491	0.008146	0.003119	0.000122	...	-0.016100	-
23	0.191437	0.007993	0.001002	0.000069	...	-0.027565	-
20	2.143477	0.021533	0.002797	0.000061	...	-0.000438	-
5	2.116438	0.042139	0.005457	0.000319	...	-0.015620	-
8	3.170330	0.047450	0.004052	0.000114	...	-0.000591	-
14	0.029755	0.000166	0.000933	0.000011	...	-0.033967	-
6	3.082627	0.030667	0.003748	0.000131	...	-0.000450	-

10 rows × 24 columns

In [37]:
```
random_cv.best_estimator_
```

Out[37]:
```
GradientBoostingRegressor(alpha=0.9, criterion='friedman_mse', init=None,
                          learning_rate=0.1, loss='lad', max_depth=2, max_features=None,
                          max_leaf_nodes=None, min_impurity_decrease=0.0,
                          min_impurity_split=None, min_samples_leaf=8,
                          min_samples_split=6, min_weight_fraction_leaf=0.0,
                          n_estimators=500, n_iter_no_change=None, presort='auto',
                          random_state=42, subsample=1.0, tol=0.0001,
                          validation_fraction=0.1, verbose=0, warm_start=False)
```

In [38]:
```
# The best gradient boosted model has the following hyperparameters:

# loss = lad
# n_estimators = 500
# max_depth = 2
# min_samples_leaf = 8
# min_samples_split = 6
# max_features = None
```

In [39]:
```
# I will focus on a single one, the number of trees in the forest (n_estimators
# By varying only one hyperparameter, we can directly observe how it affects pe
# In the case of the number of trees, we would expect to see a significant affe

# Here we will use grid search with a grid that only has the n_estimators hyper
# We will evaluate a range of trees then plot the training and testing performa
# We will fix the other hyperparameters at the best values returned from random
```

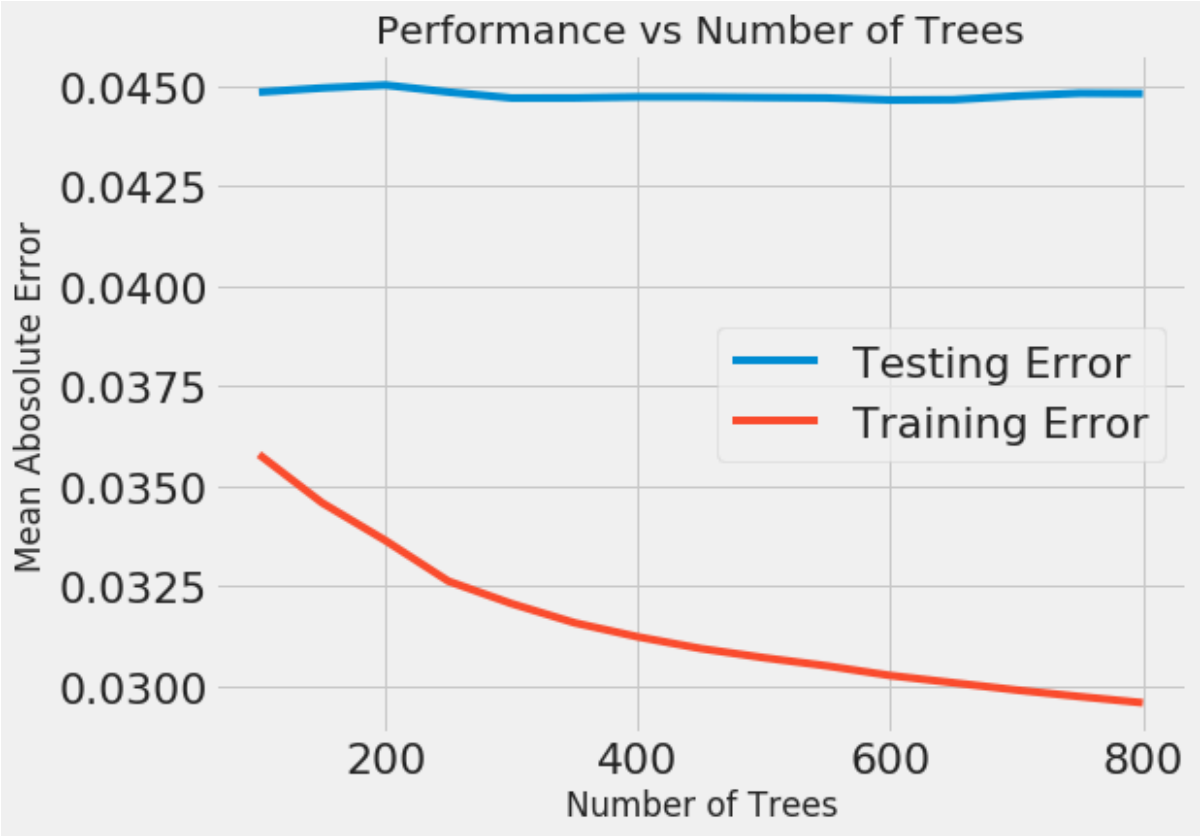
In [40]:
```
# Create a range of trees to evaluate
trees_grid = {'n_estimators': [100, 150, 200, 250, 300, 350, 400, 450, 500, 550

model = GradientBoostingRegressor(loss = 'lad', max_depth = 2,
                                  min_samples_leaf = 8,
                                  min_samples_split = 6,
                                  max_features = None,
                                  random_state = 42)

# Grid Search Object using the trees range and the random forest model
grid_search = GridSearchCV(estimator = model, param_grid=trees_grid, cv = 4,
                           scoring = 'neg_mean_absolute_error', verbose = 1,
                           n_jobs = -1, return_train_score = True)
```

In [41]:
```
# Fit the grid search
grid_search.fit(X_train, y_train)
```

```
Fitting 4 folds for each of 15 candidates, totalling 60 fits

[Parallel(n_jobs=-1)]: Using backend LokyBackend with 4 concurrent workers.
[Parallel(n_jobs=-1)]: Done  60 out of  60 | elapsed:    7.1s finished
```

Out[41]:
```
GridSearchCV(cv=4, error_score='raise-deprecating',
       estimator=GradientBoostingRegressor(alpha=0.9, criterion='friedman_ms
e', init=None,
             learning_rate=0.1, loss='lad', max_depth=2, max_features=None,
             max_leaf_nodes=None, min_impurity_decrease=0.0,
             min_impurity_split=None, min_samples_leaf=8,
             min_samp...te=42, subsample=1.0, tol=0.0001,
             validation_fraction=0.1, verbose=0, warm_start=False),
       fit_params=None, iid='warn', n_jobs=-1,
       param_grid={'n_estimators': [100, 150, 200, 250, 300, 350, 400, 450, 5
00, 550, 600, 650, 700, 750, 800]},
       pre_dispatch='2*n_jobs', refit=True, return_train_score=True,
       scoring='neg_mean_absolute_error', verbose=1)
```

In [42]:
```
# Get the results into a dataframe
results = pd.DataFrame(grid_search.cv_results_)

# Plot the training and testing error vs number of trees
figsize=(8, 8)
plt.style.use('fivethirtyeight')
plt.plot(results['param_n_estimators'], -1 * results['mean_test_score'], label
plt.plot(results['param_n_estimators'], -1 * results['mean_train_score'], label
plt.xlabel('Number of Trees'); plt.ylabel('Mean Abosolute Error'); plt.legend()
plt.title('Performance vs Number of Trees');

# There will always be a difference between the training error and testing erro
# we want to try and reduce overfitting, either by getting more training data o

# For now, we will use the model with the best performance and accept that it m
```



In [43]:
```
results.sort_values('mean_test_score', ascending = False).head(5)
```

Out[43]:

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	...	split2_train_score	split3_tra
10	0.607595	0.011356	0.002039	0.000109	...	-0.028336	-
11	0.658742	0.008958	0.002125	0.000080	...	-0.028142	-
4	0.304018	0.004849	0.001394	0.000027	...	-0.030469	-
9	0.566310	0.010390	0.002004	0.000094	...	-0.028723	-
5	0.359498	0.008033	0.001548	0.000046	...	-0.029991	-

5 rows × 19 columns

In [44]:
```
# # # Evaluate Final Model on the Test Set

# We will use the best model from hyperparameter tuning to make predictions on

# For comparison, we can also look at the performance of the default model. The

# Default model
default_model = GradientBoostingRegressor(random_state = 42)

# Select the best model
final_model = grid_search.best_estimator_

final_model
```

Out[44]:
```
GradientBoostingRegressor(alpha=0.9, criterion='friedman_mse', init=None,
                          learning_rate=0.1, loss='lad', max_depth=2, max_features=None,
                          max_leaf_nodes=None, min_impurity_decrease=0.0,
                          min_impurity_split=None, min_samples_leaf=8,
                          min_samples_split=6, min_weight_fraction_leaf=0.0,
                          n_estimators=600, n_iter_no_change=None, presort='auto',
                          random_state=42, subsample=1.0, tol=0.0001,
                          validation_fraction=0.1, verbose=0, warm_start=False)
```

In [45]:
```
%%timeit -n 1 -r 5
default_model.fit(X_train, y_train)
```

```
27.5 ms ± 1.83 ms per loop (mean ± std. dev. of 5 runs, 1 loop each)
```

In [46]:
```
%%timeit -n 1 -r 5
final_model.fit(X_train, y_train)
```

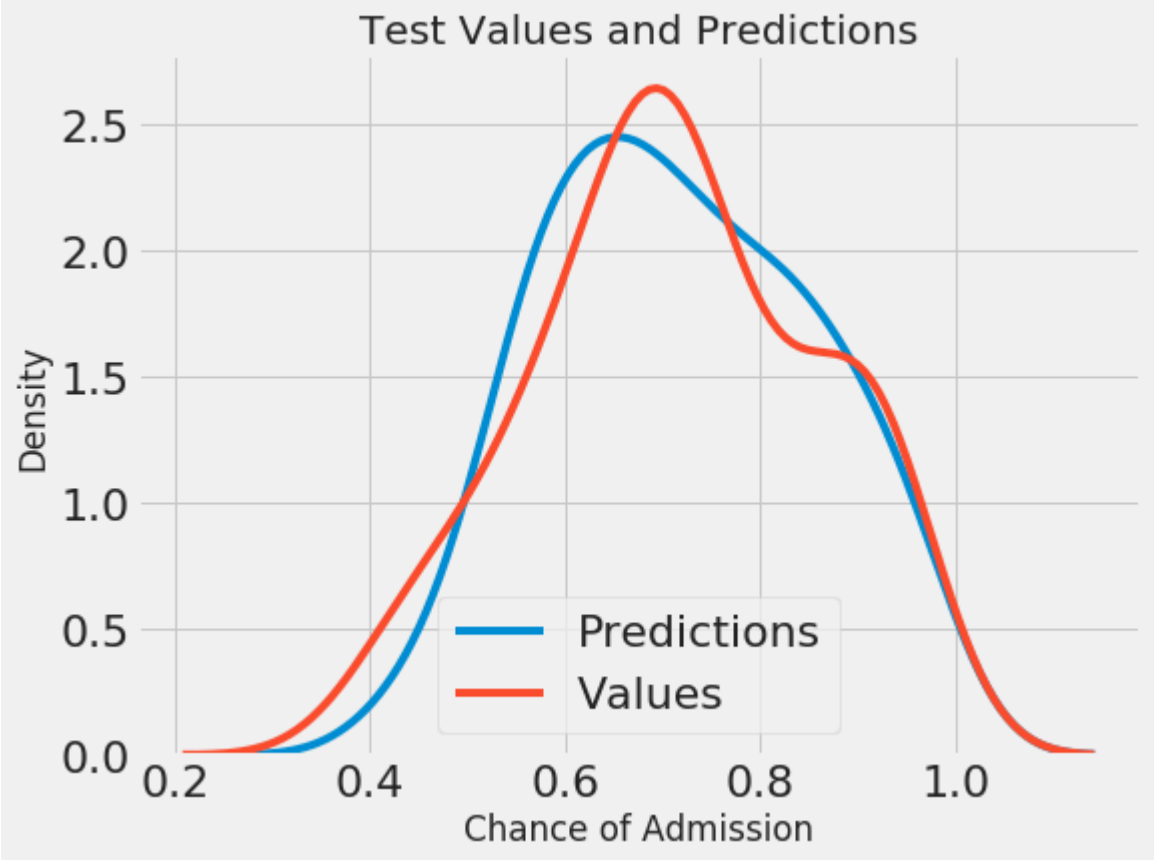
```
339 ms ± 1.35 ms per loop (mean ± std. dev. of 5 runs, 1 loop each)
```

In [47]:
```
default_pred = default_model.predict(X_test)
final_pred = final_model.predict(X_test)

print('Default model performance on the test set: MAE = %0.4f.' % mae(y_test, d
print('Final model performance on the test set:   MAE = %0.4f.' % mae(y_test, f

# The model have the very good performace!!!
```

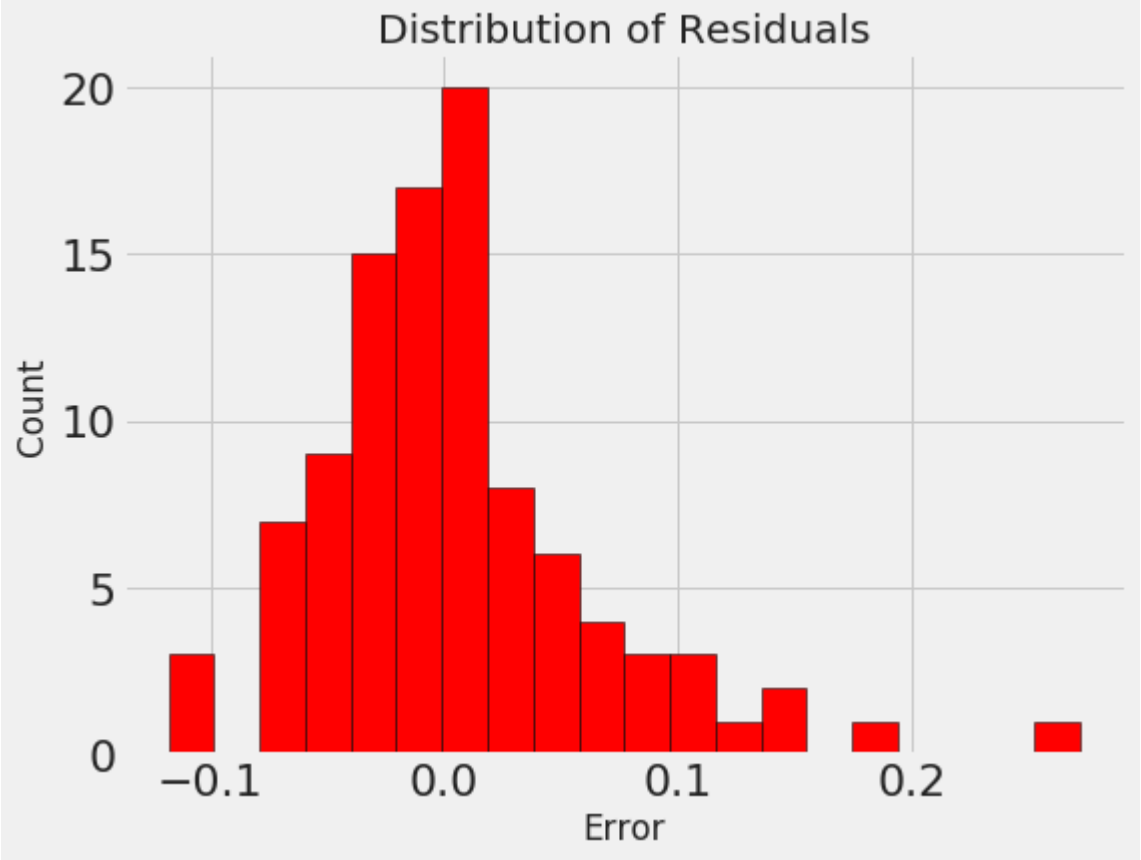
```
Default model performance on the test set: MAE = 0.0462.
Final model performance on the test set:   MAE = 0.0435.
```

In [48]:
```
# To get a sense of the predictions, we can plot the distribution of true value

# Train the model.
lr.fit(X_train, y_train)

# Make predictions and evalute.
model_pred = lr.predict(X_test)

figsize=(8, 8)

# Density plot of the final predictions and the test values.
sns.kdeplot(model_pred, label = 'Predictions')
sns.kdeplot(y_test, label = 'Values')

# Label the plot.
plt.xlabel('Chance of Admission'); plt.ylabel('Density');
plt.title('Test Values and Predictions');

# The distribution looks to be nearly the same.
```

```
/opt/conda/lib/python3.6/site-packages/scipy/stats/stats.py:1713: FutureWarni
ng: Using a non-tuple sequence for multidimensional indexing is deprecated; u
se `arr[tuple(seq)]` instead of `arr[seq]`. In the future this will be interp
reted as an array index, `arr[np.array(seq)]`, which will result either in an
error or a different result.
  return np.add.reduce(sorted[indexer] * weights, axis=axis) / sumval
```



In [49]:
```
# Another diagnostic plot is a histogram of the residuals.
# Ideally, we would hope that the residuals are normally distributed, meaning t

figsize = (6, 6)

# Calculate the residuals
residuals = model_pred - y_test

# Plot the residuals in a histogram
plt.hist(residuals, color = 'red', bins = 20,
         edgecolor = 'black')
plt.xlabel('Error'); plt.ylabel('Count')
plt.title('Distribution of Residuals');

# The residuals are close to normally disributed, with a few noticeable outlier
# These indicate errors where the model estimate was far below that of the true
```



In [50]:
```
model.fit(X_train, y_train)
```

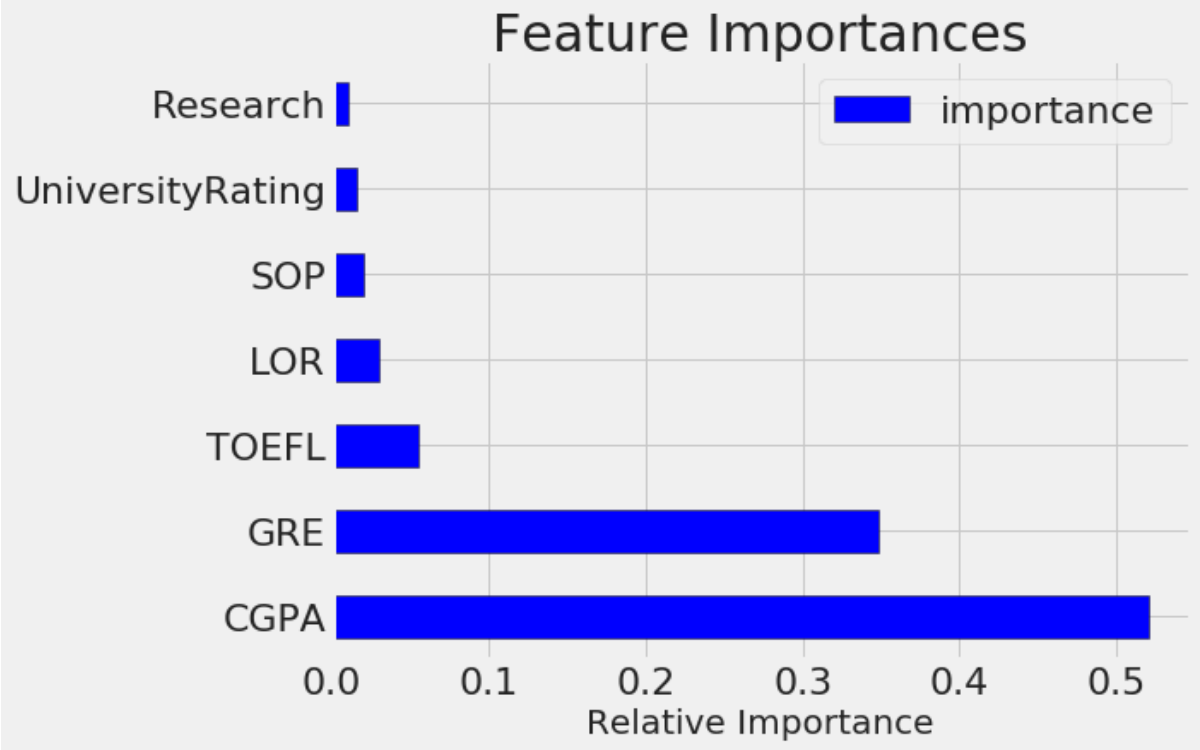
Out[50]:
```
GradientBoostingRegressor(alpha=0.9, criterion='friedman_mse', init=None,
                          learning_rate=0.1, loss='lad', max_depth=2, max_features=None,
                          max_leaf_nodes=None, min_impurity_decrease=0.0,
                          min_impurity_split=None, min_samples_leaf=8,
                          min_samples_split=6, min_weight_fraction_leaf=0.0,
                          n_estimators=100, n_iter_no_change=None, presort='auto',
                          random_state=42, subsample=1.0, tol=0.0001,
                          validation_fraction=0.1, verbose=0, warm_start=False)
```

In [51]:
```
# # # Interprete the Model

# # Feature Importances

# Extract the feature importances into a dataframe
graduate_features = graduate.drop(labels='Chance', axis=1)
feature_results = pd.DataFrame({'feature': list(graduate_features.columns),
                                'importance': model.feature_importances_})

# Show the top 10 most important
feature_results = feature_results.sort_values('importance', ascending = False).

feature_results.head(10)
```

Out[51]:

	feature	importance
0	CGPA	0.520548
1	GRE	0.348582
2	TOEFL	0.054804
3	LOR	0.030340
4	SOP	0.020161
5	UniversityRating	0.015420
6	Research	0.010145

In [52]:
```
# Let's graph the feature importances to compare visually.

figsize=(12, 10)
plt.style.use('fivethirtyeight')

# Plot the 10 most important features in a horizontal bar chart
feature_results.loc[:9, :].plot(x = 'feature', y = 'importance',
                                edgecolor = 'k',
                                kind='barh', color = 'blue');
plt.xlabel('Relative Importance', size = 20); plt.ylabel('')
plt.title('Feature Importances', size = 30);
```



In [53]:
```
# # Use Feature Importances for Feature Selection

# Let's try using only the 10 most important features in the linear regression
# We can also limit to these features and re-evaluate the random forest.

# Extract the names of the most important features
most_important_features = feature_results['feature'][:10]

# Find the index that corresponds to each feature name
indices = [list(graduate_features.columns).index(x) for x in most_important_fea

# Keep only the most important features
X_train_reduced = X_train[:, indices]
X_test_reduced = X_test[:, indices]

print('Most important training features shape: ', X_train_reduced.shape)
print('Most important testing  features shape: ', X_test_reduced.shape)
```

```
Most important training features shape:  (400, 7)
Most important testing  features shape:  (100, 7)
```

In [54]:
```
lr = LinearRegression()

# Fit on full set of features
lr.fit(X_train, y_train)
lr_full_pred = lr.predict(X_test)

# Fit on reduced set of features
lr.fit(X_train_reduced, y_train)
lr_reduced_pred = lr.predict(X_test_reduced)

# Display results
print('Linear Regression Full Results: MAE =    %0.4f.' % mae(y_test, lr_full_p
print('Linear Regression Reduced Results: MAE = %0.4f.' % mae(y_test, lr_reduce

# Well, reducing the features did not improve the linear regression results!
# It turns out that the extra information in the features with low importance a
```

```
Linear Regression Full Results: MAE =    0.0427.
Linear Regression Reduced Results: MAE = 0.0427.
```

In [55]:
```
# Let's look at using the reduced set of features in the gradient boosted regre

# Create the model with the same hyperparamters
model_reduced = GradientBoostingRegressor(loss='lad', max_depth=2, max_features
                                  min_samples_leaf=8, min_samples_split=6,
                                  n_estimators=800, random_state=42)

# Fit and test on the reduced set of features
model_reduced.fit(X_train_reduced, y_train)
model_reduced_pred = model_reduced.predict(X_test_reduced)

print('Gradient Boosted Reduced Results: MAE = %0.4f' % mae(y_test, model_reduc

# The model results are slightly worse with the reduced set of features and we
```

```
Gradient Boosted Reduced Results: MAE = 0.0429
```

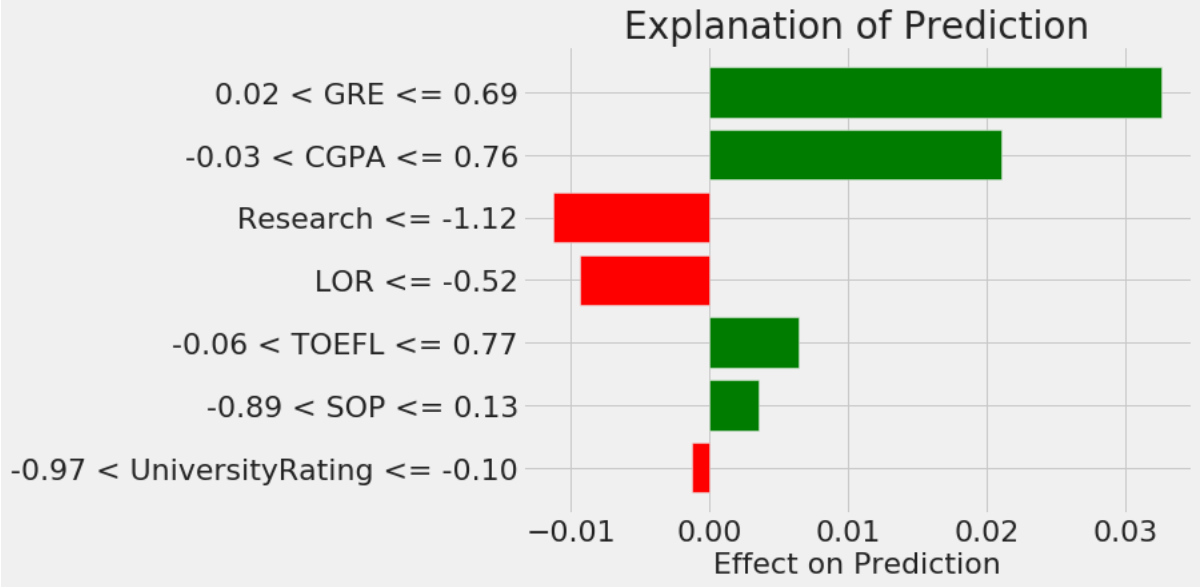
In [60]:
```
# # Locally Interpretable Model-agnostic Explanations

# We will look at using LIME to explain individual predictions made the by the
#LIME is a relatively new effort aimed at showing how a machine learning model

# We will look at trying to explain the predictions on an example the model get
#We will restrict ourselves to using the reduced set of 10 features to aid inte
#The model trained on the 10 most important features is slightly less accurate,
```

In [56]:
```
# Find the residuals
residuals = abs(model_reduced_pred - y_test)

# Exact the worst and best prediction
wrong = X_test_reduced[np.argmax(residuals), :]
right = X_test_reduced[np.argmin(residuals), :]
```

In [57]:
```
# Create a lime explainer object
# LIME for explaining predictions
import lime
import lime.lime_tabular

explainer = lime.lime_tabular.LimeTabularExplainer(training_data = X_train_redu
                                                   mode = 'regression',
                                                   training_labels = y_train,
                                                   feature_names = list(most_im
```

In [58]:
```
# Display the predicted and true value for the wrong instance
print('Prediction: %0.4f' % model_reduced.predict(wrong.reshape(1, -1)))
print('Actual Value: %0.4f' % y_test[np.argmax(residuals)])

# Explanation for wrong prediction
wrong_exp = explainer.explain_instance(data_row = wrong,
                                       predict_fn = model_reduced.predict)

# Plot the prediction explaination
wrong_exp.as_pyplot_figure();
plt.title('Explanation of Prediction', size = 28);
plt.xlabel('Effect on Prediction', size = 22);

# In this example, our gradient boosted model predicted a score of 0.7288 and t

# The plot from LIME is showing us the contribution to the final prediction fro

# We can see that the GRE singificantly increased the prediction when we compar
# The Research on the other hand, decreased the prediction when we comparing wi
```

```
Prediction: 0.7288
Actual Value: 0.4500
```



In [34]:
```
# Now we can go through the same process with a prediction the model got correc

# Display the predicted and true value for the wrong instance
print('Prediction: %0.4f' % model_reduced.predict(right.reshape(1, -1)))
print('Actual Value: %0.4f' % y_test[np.argmin(residuals)])

# Explanation for wrong prediction
right_exp = explainer.explain_instance(right, model_reduced.predict, num_featur
right_exp.as_pyplot_figure();
plt.title('Explanation of Prediction', size = 28);
plt.xlabel('Effect on Prediction', size = 22);

# The correct value for this case was 0.8899 which our gradient boosted model g

# The plot from LIME again shows the contribution to the prediciton of each of

# Observing break down plots like these allow us to get an idea of how the mode
# This is probably most valuable for cases where the model is off by a large am
# to improve predictions for next time. The examples where the model is off the
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[34], line 4
      1 # Now we can go through the same process with a prediction the model
got correct.
      2
      3 # Display the predicted and true value for the wrong instance
----> 4 print('Prediction: %0.4f' % model_reduced.predict(right.reshape(1, -
1)))
      5 print('Actual Value: %0.4f' % y_test[np.argmin(residuals)])
      7 # Explanation for wrong prediction

NameError: name 'model_reduced' is not defined
```

In [61]:
```
# A process such as this where we try to work with the machine learning algorit
# and completely trusting them! Although LIME is not perfect, it represents a s
```

In [62]:
```
# Good job with this project!
# See you in the next one!!!
```

In [ ]:
```
# I will use in this Kernel the step-by-step process of Will Koehrsen.
# I won't use everything, but most of them.
# This project at in GitHub repository: https://github.com/WillKoehrsen/machine
```