Applied Machine Learning

HW2 :Task 1

Maxime TCHIBOZO (mt3390)

Disclaimer : For a reason I have not been able to identify, the pipelines I created in this task tended to generate ConvergenceWarnings. I have opted to ignore these warnings as the results output by the functions were good.

In [1]:
```
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
from numpy.random import RandomState
from sklearn.datasets import fetch_openml
from sklearn.model_selection import train_test_split, GridSearchCV, KFold
from sklearn.preprocessing import OneHotEncoder, StandardScaler
from sklearn.linear_model import LogisticRegression, Ridge
from sklearn.compose import make_column_transformer
from sklearn.pipeline import make_pipeline, Pipeline
from sklearn.model_selection import cross_val_score
from sklearn.svm import LinearSVC
from sklearn.neighbors import KNeighborsClassifier

import warnings
```

In [2]:
```
#collect the dataset
credit_g_dict = fetch_openml("credit-g")
```

1.1 Categorical Features vs. Continuous Features

In [4]:
```
print(credit_g_dict.feature_names)
```

```
['checking_status', 'duration', 'credit_history', 'purpose', 'credit_amount',
'savings_status', 'employment', 'installment_commitment', 'personal_status', '
other_parties', 'residence_since', 'property_magnitude', 'age', 'other_payment
_plans', 'housing', 'existing_credits', 'job', 'num_dependents', 'own_telephon
e', 'foreign_worker']
```

In total, we have 20 features (excluding the target). Some of the categorical features are detailed by the "categories" key of the credit_g dictionary. Upon further analysis, we observe that there are additional categorical variables : installment_commitment, residence_since ,existing_credits ,num_dependents.

In [5]:
```
print("The Categorical Features are : "+str(list(credit_g_dict["categories"].key
s())+["installment_commitment", "residence_since" ,"existing_credits" ,"num_depe
ndents"]))
```

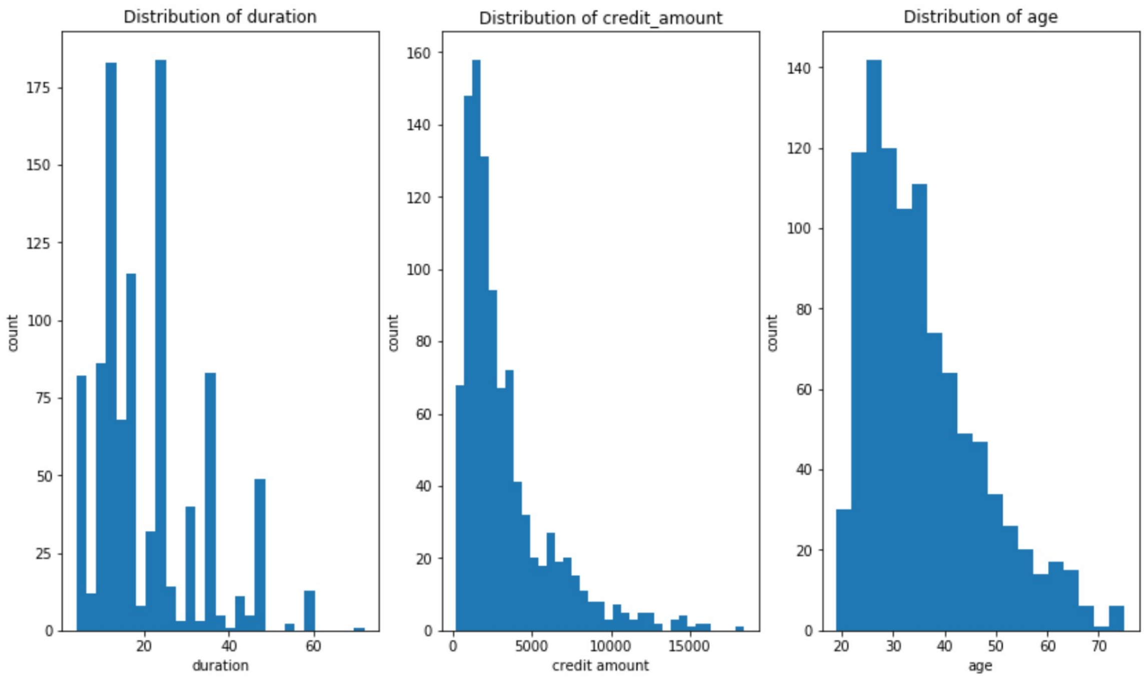
```
The Categorical Features are : ['checking_status', 'credit_history', 'purpose
', 'savings_status', 'employment', 'personal_status', 'other_parties', 'proper
ty_magnitude', 'other_payment_plans', 'housing', 'job', 'own_telephone', 'fore
ign_worker', 'installment_commitment', 'residence_since', 'existing_credits',
'num_dependents']
```

We can consider that the following features are continuous : duration, credit_amount and age

1.2 Distributions of Continuous Features and Target

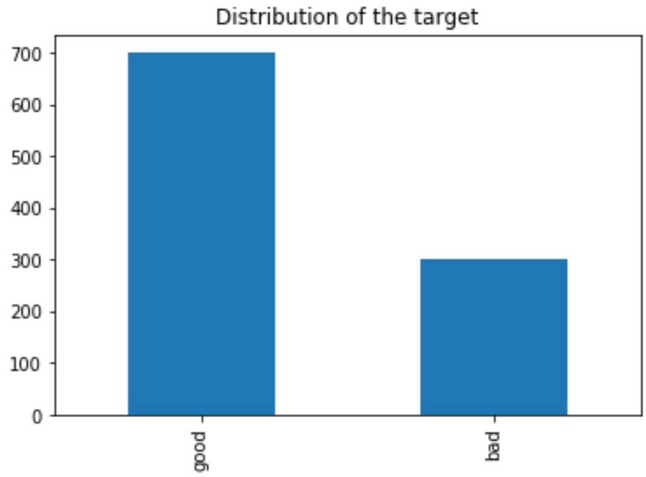
In [6]:
```
#create feature and target data frames
df_X = pd.DataFrame(credit_g_dict.data,columns=credit_g_dict.feature_names)
df_y = pd.DataFrame(credit_g_dict.target,columns=credit_g_dict.target_names)
```

In [7]:
```
#distribution of the continuous variables
fig, ax = plt.subplots(1,3,figsize=(14,8))

ax[0].hist(df_X["duration"],bins="auto",)
ax[1].hist(df_X["credit_amount"],bins="auto")
ax[2].hist(df_X["age"],bins="auto")

ax[0].set_xlabel("duration")
ax[1].set_xlabel("credit amount")
ax[2].set_xlabel("age")

ax[0].set_ylabel("count")
ax[1].set_ylabel("count")
ax[2].set_ylabel("count")

ax[0].title.set_text("Distribution of duration")
ax[1].title.set_text("Distribution of credit_amount")
ax[2].title.set_text("Distribution of age")
```



In [8]:
```
#distribution of the target
df_y['class'].value_counts().plot(kind='bar')
plt.title("Distribution of the target")
```

Out[8]: Text(0.5, 1.0, 'Distribution of the target')



1.3 Preprocessing of the features

In [9]:
```
#We encode the target to 0-1 values
df_y["IsGood"] = df_y["class"].apply(lambda x: int(x=="good"))

#We convert the categorical variables to objects
#To do so we consider all columns other than age, duration and credit_amount
df = df_X[df_X.columns.difference(["age","duration","credit_amount"])].astype("o
bject")

#We dummy encode the categorical variables and save them to df
df = pd.get_dummies(df)

#We scale the continuous variables and append them to df
scaler = StandardScaler()
scaler.fit(df_X[["age","duration","credit_amount"]])
df = pd.concat([df, pd.DataFrame(scaler.transform(df_X[["age","duration","credit
_amount"]]),columns=["age","duration","credit_amount"])
], axis=1)
```

In [10]:
```
np.shape(df)
```

Out[10]: (1000, 71)

Scaling the continuous variables and encoding the categorical variables now means our preprocessed dataframe, df, has 71 features.

Let us note that we also encoded the values of the target to be 1 if the class is "good" and 0 otherwise (stored in df_y["IsGood"]).

In [11]:
```
#Initial Logistic Regression Model

X_train, X_test, y_train, y_test = train_test_split(df,df_y["IsGood"])

#We evaluate the model using a training/validation split
X_train_2, X_validate, y_train_2, y_validate = train_test_split(X_train,y_train)
model = LogisticRegression().fit(X_train_2,y_train_2)
model.score(X_validate,y_validate)
```

Out[11]:
```
0.7446808510638298
```

The logistic regression model without using pipelines yields 74.5% accuracy on the validation set.

1.4 Using Pipelines

We fit the first models without scaling the continuous features. We Dummy encode the categorical variables.

In [13]:
```
#We compare the classifiers without scaling the continuous features
warnings.filterwarnings('ignore')

df = df_X[df_X.columns.difference(["age","duration","credit_amount"])].astype("o
bject")
df = pd.concat([df,df_X[["age","duration","credit_amount"]]],axis=1)

categorical = df.dtypes == object

preprocess = make_column_transformer((OneHotEncoder(), categorical),("passthroug
h",~categorical))


X_train, X_test, y_train, y_test = train_test_split(df,df_y["IsGood"])

model_log_reg = make_pipeline(preprocess, LogisticRegression())
model_lin_SVC = make_pipeline(preprocess, LinearSVC())
model_KNN = make_pipeline(preprocess,KNeighborsClassifier())

scores_log_reg = np.mean(cross_val_score(model_log_reg, X_train, y_train))
scores_lin_SVC = np.mean(cross_val_score(model_lin_SVC,X_train,y_train))
scores_KNN = np.mean(cross_val_score(model_KNN,X_train,y_train))

print("Logistic Regression with Pipeline Score : "+str(scores_log_reg))
print("Linear SVC with Pipeline Score : "+str(scores_lin_SVC))
print("K-Nearest Neighbors with Pipeline Score : "+str(scores_KNN))
```

```
Logistic Regression with Pipeline Score : 0.7266666666666666
Linear SVC with Pipeline Score : 0.6973333333333332
K-Nearest Neighbors with Pipeline Score : 0.6533333333333333
```

In some cases, the returned scores are nan. This seems to indicate that we should scale our continuous variables if we want to obtain consistent results.

Let us see if the score on the dataset with scaled continuous features is better than for unscaled data(this would confirm our decision to scale).

We compare these results to those obtained when we scale the continuous features.

In [12]:
```
warnings.filterwarnings('ignore')
df = df_X[df_X.columns.difference(["age","duration","credit_amount"])].astype("o
bject")
df = pd.concat([df,df_X[["age","duration","credit_amount"]]],axis=1)

categorical = df.dtypes == object

preprocess = make_column_transformer((StandardScaler(), ~categorical),(OneHotEnc
oder(), categorical))


X_train, X_test, y_train, y_test = train_test_split(df,df_y["IsGood"])
X_train_2, X_validate, y_train_2, y_validate = train_test_split(X_train,y_train)

model_log_reg = make_pipeline(preprocess, LogisticRegression())
model_lin_SVC = make_pipeline(preprocess, LinearSVC())
model_KNN = make_pipeline(preprocess,KNeighborsClassifier())

scores_log_reg = np.mean(cross_val_score(model_log_reg, X_train, y_train))
scores_lin_SVC = np.mean(cross_val_score(model_lin_SVC,X_train,y_train))
scores_KNN = np.mean(cross_val_score(model_KNN,X_train,y_train))

print("Logistic Regression with Pipeline Score : "+str(scores_log_reg))
print("Linear SVC with Pipeline Score : "+str(scores_lin_SVC))
print("K-Nearest Neighbors with Pipeline Score : "+str(scores_KNN))
```

```
Logistic Regression with Pipeline Score : 0.7573333333333334
Linear SVC with Pipeline Score : 0.756
K-Nearest Neighbors with Pipeline Score : 0.74
```

We observe that in general, scaling the continuous variables improves accuracy (a few percentage points improvement). From now on, we will consider the data where the continuous variables are scaled.

In [13]:
```
#We scale the continuous variables, Dummy encode the categorical variables, and
train-test split
df = df_X[df_X.columns.difference(["age","duration","credit_amount"])].astype("o
bject")
df = pd.concat([df,df_X[["age","duration","credit_amount"]]],axis=1)

categorical = df.dtypes == object

preprocess = make_column_transformer((OneHotEncoder(), categorical),(StandardSca
ler(), ~categorical))

df = make_pipeline(preprocess).fit_transform(df)

X_train,X_test,y_train,y_test = train_test_split(df,df_y["IsGood"])

#We convert X_train and X_test from sparse matrix to array
X_train = X_train.toarray()
X_test = X_test.toarray()
```

1.5 Parameter Tuning

In [14]:
```
warnings.filterwarnings('ignore')
pipe = Pipeline([("regressor",LogisticRegression())])
param_grid = [{'regressor': [LogisticRegression()],
                'regressor__C':np.logspace(-3,3,10)},
               {'regressor': [LinearSVC()],
                'regressor__C':np.logspace(-3,3,10)},
               {'regressor': [KNeighborsClassifier()],
               'regressor__n_neighbors':range(1,10)
              }]

grid = GridSearchCV(pipe, param_grid,cv=5)
grid.fit(X_train,y_train)

print("Best model based on training : "+ str(grid.best_params_))
print("Score of best model on training : "+str(grid.score(X_train,y_train)))
```

```
Best model based on training : {'regressor': LogisticRegression(C=2.1544346900
31882, class_weight=None, dual=False,
                   fit_intercept=True, intercept_scaling=1, l1_ratio=None,
                   max_iter=100, multi_class='auto', n_jobs=None, penalty='l2
',
                   random_state=None, solver='lbfgs', tol=0.0001, verbose=0,
                   warm_start=False), 'regressor__C': 2.154434690031882}
Score of best model on training : 0.7946666666666666
```

Parameter Tuning improves results by a few percentage points. The model which best performs on training data is sometimes Logistic Regression and sometimes LinearSVC. This seems to indicate that both methods reach similar optima. In this run, it was a Logistic Regression model with C=2.15. Let us see how this model performs on the test set.

In [15]:
```
print("Test score : "+ str(grid.score(X_test,y_test)))
```

```
Test score : 0.744
```

The test score is similar to the validation scores of the previous models. This seems to indicate that the model has not overfitted.

In [16]:

```
#Performance as function of Parameters for Logistic Regression, LinearSVC and KN
N
warnings.filterwarnings('ignore')
C = np.logspace(-3,3,10)
n_neighbors = range(1,10)

LR_test_score_grid = [LogisticRegression(C=C[i]).fit(X_train,y_train).score(X_te
st,y_test) for i in range(len(C))]
SVC_test_score_grid = [LinearSVC(C=C[i]).fit(X_train,y_train).score(X_test,y_tes
t) for i in range(len(C))]
KNN_test_score_grid = [KNeighborsClassifier(n_neighbors=n_neighbors[i]).fit(X_tr
ain,y_train).score(X_test,y_test) for i in range(len(n_neighbors))]

fig, ax = plt.subplots(1,3,figsize=(10,7))

ax[0].plot(C,LR_test_score_grid,label="Score")
ax[0].set_xscale("log")
ax[0].set_xlabel("C")
ax[0].set_ylabel("score")
ax[0].title.set_text("Test Score of Logistic Regression")
ax[0].legend()

ax[1].plot(C,SVC_test_score_grid,label="Score")
ax[1].set_xscale("log")
ax[1].set_xlabel("C")
ax[1].set_ylabel("score")
ax[1].title.set_text("Test Score of Linear SVC")
ax[1].legend()

ax[2].plot(n_neighbors,KNN_test_score_grid,label="Score")
ax[2].set_xlabel("n_neighbors")
ax[2].set_ylabel("score")
ax[2].title.set_text("Test Score of K-Nearest Neighbors")
ax[2].legend()
```

Out[16]: <matplotlib.legend.Legend at 0x280742a9828>

The performance graphs confirm that Logistic Regression models are the most reliable of the three classification methods, as the test scores are consistently above 70% even when the parameters change. LinearSVC tends to have worse results for large values of C. K-Nearest Neighbors results are lower in general than for the other two methods.

1.6 Cross Validation without Stratification and with Shuffling

In [17]:
```
# KFold Cross-validation with Shuffling
warnings.filterwarnings('ignore')
cv = KFold(5,shuffle=True)

pipe = Pipeline([('regressor',LogisticRegression())])

param_grid = [{'regressor': [LogisticRegression()],
               'regressor__C':np.logspace(-3,3,10)},
              {'regressor': [LinearSVC()],
               'regressor__C':np.logspace(-3,3,10)},
              {'regressor': [KNeighborsClassifier()],
              'regressor__n_neighbors':range(1,10)
             }]

grid = GridSearchCV(pipe, param_grid,cv=cv,)
grid.fit(X_train,y_train)

print("Best model based on training : "+ str(grid.best_params_))
print("Score of best model on test : "+str(grid.score(X_test,y_test)))
```

```
Best model based on training : {'regressor': LogisticRegression(C=0.4641588833
6127775, class_weight=None, dual=False,
                   fit_intercept=True, intercept_scaling=1, l1_ratio=None,
                   max_iter=100, multi_class='auto', n_jobs=None, penalty='l2
',
                   random_state=None, solver='lbfgs', tol=0.0001, verbose=0,
                   warm_start=False), 'regressor__C': 0.46415888336127775}
Score of best model on test : 0.732
```

When we use shuffling and KFold, the best model on this run was LogisticRegression, but the value of the parameter found changed.

In [18]:
```
# KFold Cross-Validation with random seed Shuffling
warnings.filterwarnings('ignore')
cv = KFold(5,shuffle=True,random_state=RandomState(1))

pipe = Pipeline([('regressor',LogisticRegression())])

param_grid = [{'regressor': [LogisticRegression()],
               'regressor__C':np.logspace(-3,3,10)},
              {'regressor': [LinearSVC()],
               'regressor__C':np.logspace(-3,3,10)},
              {'regressor': [KNeighborsClassifier()],
              'regressor__n_neighbors':range(1,10)
             }]

grid = GridSearchCV(pipe, param_grid,cv=cv,)
grid.fit(X_train,y_train)

print("Best model based on training : "+ str(grid.best_params_))
print("Score of best model on test : "+str(grid.score(X_test,y_test)))
```

```
Best model based on training : {'regressor': LinearSVC(C=0.021544346900318832,
class_weight=None, dual=True,
          fit_intercept=True, intercept_scaling=1, loss='squared_hinge',
          max_iter=1000, multi_class='ovr', penalty='l2', random_state=None,
          tol=0.0001, verbose=0), 'regressor__C': 0.021544346900318832}
Score of best model on test : 0.72
```

Changing the Random Seed has once again changed the optimal estimator. The best estimator is now a LinearSVC model with C=0.022(results might change if you run the code again).

In [19]:
```
# KFold Cross-Validation with Shuffling, random seed and random state train test
split
warnings.filterwarnings('ignore')
X_train, X_test, y_train, y_test = train_test_split(df,df_y["IsGood"],random_sta
te=RandomState(1))

#We convert X_train and X_test from sparse matrix to array
X_train = X_train.toarray()
X_test = X_test.toarray()

cv = KFold(5,shuffle=True,random_state=RandomState(1))

pipe = Pipeline([('regressor',LogisticRegression())])

param_grid = [{'regressor': [LogisticRegression()],
               'regressor__C':np.logspace(-3,3,10)},
              {'regressor': [LinearSVC()],
               'regressor__C':np.logspace(-3,3,10)},
              {'regressor': [KNeighborsClassifier()],
              'regressor__n_neighbors':range(1,10)
             }]

grid = GridSearchCV(pipe, param_grid,cv=cv,)
grid.fit(X_train,y_train)

print("Best model based on training : "+ str(grid.best_params_))
print("Score of best model on test : "+str(grid.score(X_test,y_test)))
```

```
Best model based on training : {'regressor': LinearSVC(C=0.021544346900318832,
class_weight=None, dual=True,
          fit_intercept=True, intercept_scaling=1, loss='squared_hinge',
          max_iter=1000, multi_class='ovr', penalty='l2', random_state=None,
          tol=0.0001, verbose=0), 'regressor__C': 0.021544346900318832}
Score of best model on test : 0.76
```

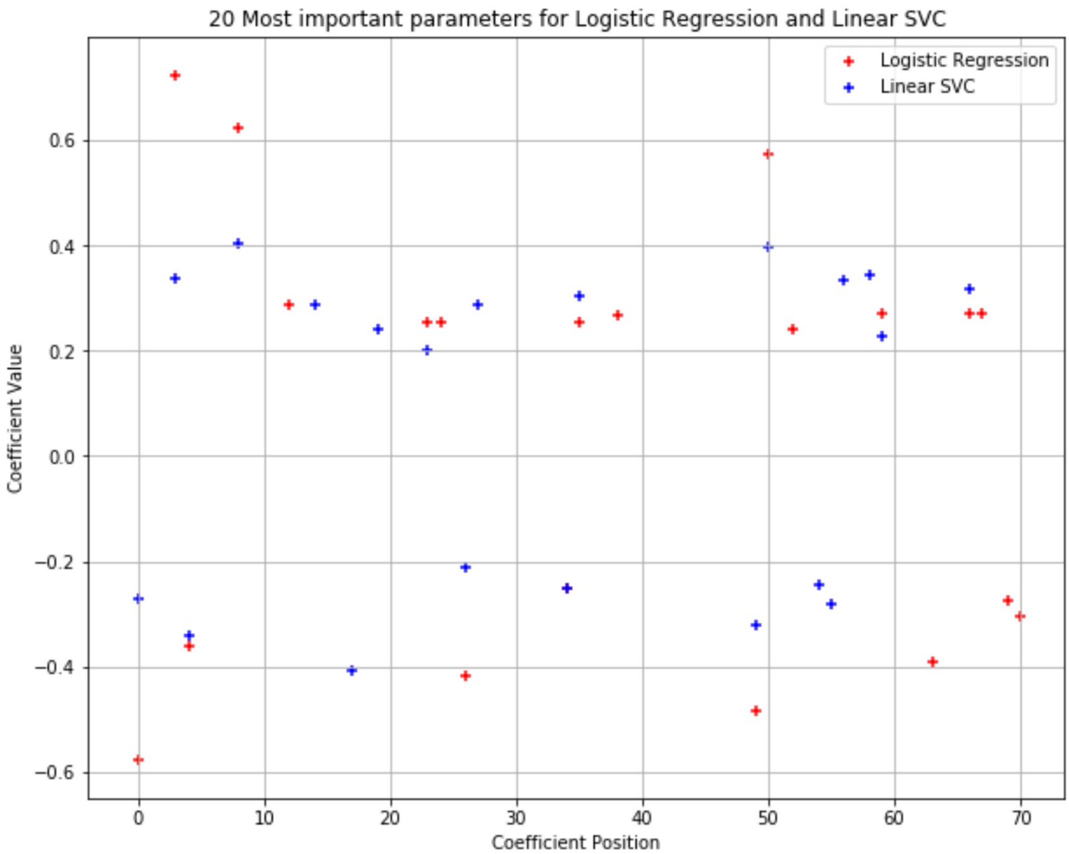
Changing the random state of the split into training and test and repeating this process for the train/validation split did not change the value of the parameter or the estimator in this instance. The best model is still a LinearSVC model. It is worth noting that for some runs, the best estimator is a LogisticRegression, whose parameters also change when we introduce shuffling, KFold and RandomStates.

1.7 Visualizing main coefficients

In [20]:
```
#We first select satisfactory parameters for LogisticRegression
warnings.filterwarnings('ignore')
pipe = Pipeline([('regressor',LogisticRegression())])

param_grid = [{'regressor': [LogisticRegression()],
               'regressor__C':np.logspace(-3,3,10)}]

grid = GridSearchCV(pipe, param_grid)
grid.fit(X_train,y_train)

model_LR = LogisticRegression(C = grid.best_params_["regressor__C"]).fit(X_trai
n,y_train)

#We select satisfactory parameters for LinearSVC

pipe = Pipeline([('regressor',LinearSVC())])

param_grid = [{'regressor': [LinearSVC()],
               'regressor__C':np.logspace(-3,3,10)}]

grid = GridSearchCV(pipe, param_grid)
grid.fit(X_train,y_train)

model_SVC = LinearSVC(C = grid.best_params_["regressor__C"]).fit(X_train,y_trai
n)
```

In [21]:
```
#We select the 20 most important coefficients (i.e those with largest absolute v
alue) for Logistic Regression and LinearSVC
coef_LR = np.absolute(model_LR.coef_)
coef_LR = np.argsort(coef_LR) #sorts from smallest to largest
coef_LR = np.flip(coef_LR)[0][:20] #selects 20 most important coefs

coef_SVC = np.absolute(model_SVC.coef_)
coef_SVC = np.argsort(coef_SVC)
coef_SVC = np.flip(coef_SVC)[0][:20]

plt.figure(figsize=(10,8))
plt.scatter(coef_LR,[model_LR.coef_[0][i] for i in coef_LR],marker="+",c="r",lab
el="Logistic Regression")
plt.scatter(coef_SVC,[model_SVC.coef_[0][i] for i in coef_SVC],marker="+",c="b",
label="Linear SVC")
plt.xlabel("Coefficient Position")
plt.ylabel("Coefficient Value")
plt.title("20 Most important parameters for Logistic Regression and Linear SVC")
plt.grid()
plt.legend(loc="upper right")
```

Out[21]: <matplotlib.legend.Legend at 0x280740128d0>



In [22]:
```
print(np.sort(coef_LR))
print(np.sort(coef_SVC))
```

```
[ 0  3  4  8 12 23 24 26 34 35 38 49 50 52 59 63 66 67 69 70]
[ 0  3  4  8 14 17 19 23 26 27 34 35 49 50 54 55 56 58 59 66]
```

The 20 most important parameters for Linear Regression and for Linear SVC are almost identical (even though their magnitudes are different).

Task 2 : Sydney Dataset

Maxime TCHIBOZO

In [246]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import category_encoders as ce

from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.compose import make_column_transformer
from sklearn.pipeline import make_pipeline, Pipeline
from sklearn.linear_model import LinearRegression, Ridge, Lasso, ElasticNet
from sklearn.model_selection import cross_val_score
from sklearn.preprocessing import OneHotEncoder, StandardScaler
from sklearn.impute import SimpleImputer

import warnings
```

Disclaimer : to ensure reproducibility of results, please make sure the csv file containing the dataset is named "sydney-data.csv"

In [247]:
```
df = pd.read_csv("sydney-data.csv")
print("The shape of the full dataframe is : "+str(np.shape(df)))
```

```
The shape of the full dataframe is : (4600, 18)
```

2.1 Drop Invalid rows, determine continuous and categorical features

In [248]:
```
#We drop the rows whose price is equal to 0
df = df[df["price"] != 0]

#We remove the date column
df = df.drop(["date"],axis=1)

print("The shape of the dataframe when we remove date and rows with price = 0 i
s : "+str(np.shape(df)))
df.head()
```

```
The shape of the dataframe when we remove date and rows with price = 0 is : (4
551, 17)
```

Out[248]:

	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors	waterfront	view	condition	sqft_above	s
0	313000.0	3.0	1.50	1340	7912	1.5	0	0	3	1340	
1	2384000.0	5.0	2.50	3650	9050	2.0	0	4	5	3370	
2	342000.0	3.0	2.00	1930	11947	1.0	0	0	4	1930	
3	420000.0	3.0	2.25	2000	8030	1.0	0	0	4	1000	
4	550000.0	4.0	2.50	1940	10500	1.0	0	0	4	1140	

The continuous features are : sqft_living, sft_lot,sqft_above, sqft_basement.

The target: price is continuous.

The categorical features are : bedrooms, bathrooms, floors, waterfront, view, condition, yr_built, yr_renovated, street, city, and statezip.

In [249]:
```
df["country"].unique()
```

Out[249]:
```
array(['USA'], dtype=object)
```

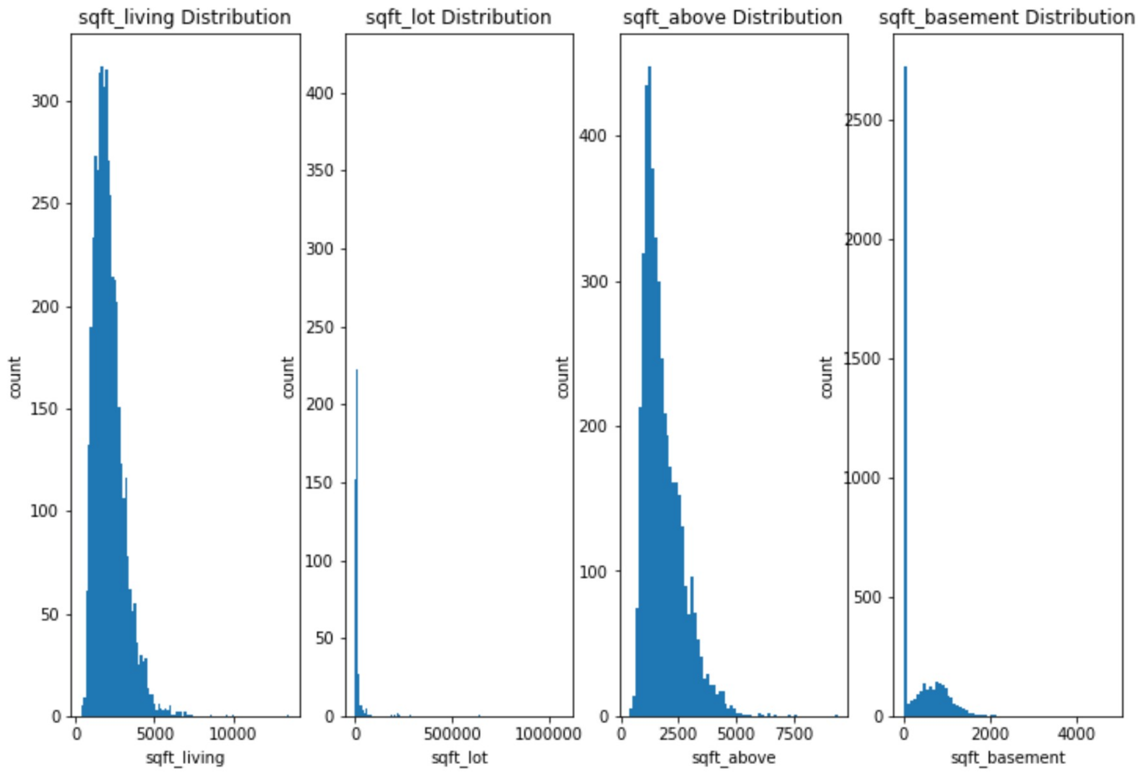
All of the data comes from the same country : USA, so we can remove the country column (it does not add any information on price).

In [250]:
```
df = df.drop(["country"],axis=1)
```

2.2 Distribution of continuous features, Distribution of target

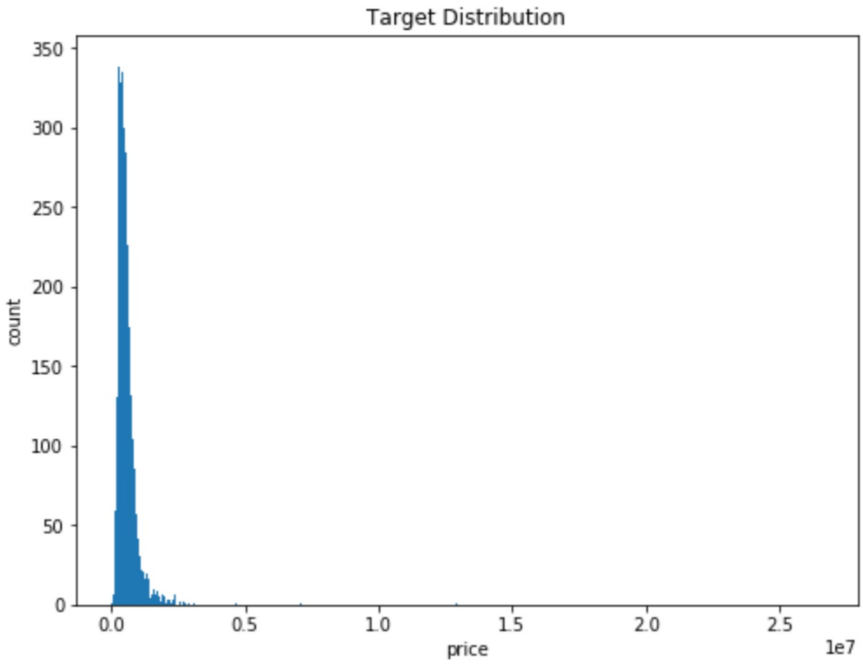
In [251]:
```
#Distribution of the continuous features
fig, ax = plt.subplots(1,4,figsize=(12,8))

ax[0].hist(df["sqft_living"],bins="auto")
ax[0].title.set_text("sqft_living Distribution")
ax[0].set_xlabel("sqft_living")
ax[0].set_ylabel("count")

ax[1].hist(df["sqft_lot"],bins="auto")
ax[1].title.set_text("sqft_lot Distribution")
ax[1].set_xlabel("sqft_lot")
ax[1].set_ylabel("count")

ax[2].hist(df["sqft_above"],bins="auto")
ax[2].title.set_text("sqft_above Distribution")
ax[2].set_xlabel("sqft_above")
ax[2].set_ylabel("count")

ax[3].hist(df["sqft_basement"],bins="auto")
ax[3].title.set_text("sqft_basement Distribution")
ax[3].set_xlabel("sqft_basement")
ax[3].set_ylabel("count")
```

Out[251]:
```
Text(0, 0.5, 'count')
```



In [252]:
```
#Distribution of the target
plt.figure(figsize=(8,6))
plt.hist(df["price"],bins="auto")
plt.title("Target Distribution")
plt.xlabel("price")
plt.ylabel("count")
```

Out[252]: Text(0, 0.5, 'count')

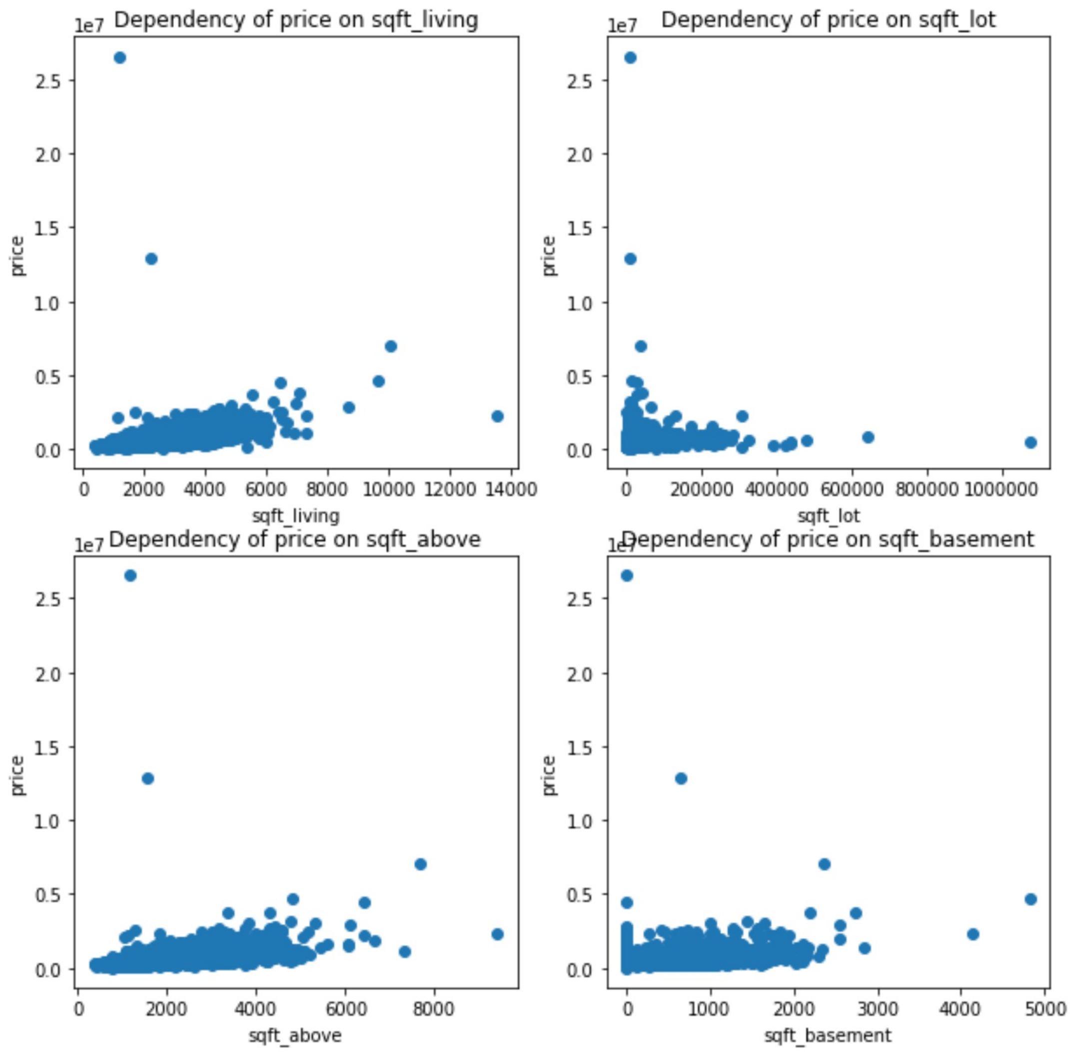


Judging by the empirical distribution of price, and the four categorical features, there are outliers which might have to be removed in order for our analysis to be accurate. It is worth noting that we have already removed rows with price=0. Additionally, the sqft_lot and sqft_basement features have peaks at 0. Going forward, we will remove the outliers on price.

2.3 Target-Feature pairwise plots

In [253]:
```
#We plot the dependency of plot
fig, ax = plt.subplots(2,2,figsize=(10,10))

ax[0,0].scatter(df["sqft_living"],df["price"])
ax[0,0].title.set_text("Dependency of price on sqft_living")
ax[0,0].set_xlabel("sqft_living")
ax[0,0].set_ylabel("price")

ax[0,1].scatter(df["sqft_lot"],df["price"])
ax[0,1].title.set_text("Dependency of price on sqft_lot")
ax[0,1].set_xlabel("sqft_lot")
ax[0,1].set_ylabel("price")

ax[1,0].scatter(df["sqft_above"],df["price"])
ax[1,0].title.set_text("Dependency of price on sqft_above")
ax[1,0].set_xlabel("sqft_above")
ax[1,0].set_ylabel("price")

ax[1,1].scatter(df["sqft_basement"],df["price"])
ax[1,1].title.set_text("Dependency of price on sqft_basement")
ax[1,1].set_xlabel("sqft_basement")
ax[1,1].set_ylabel("price")
```

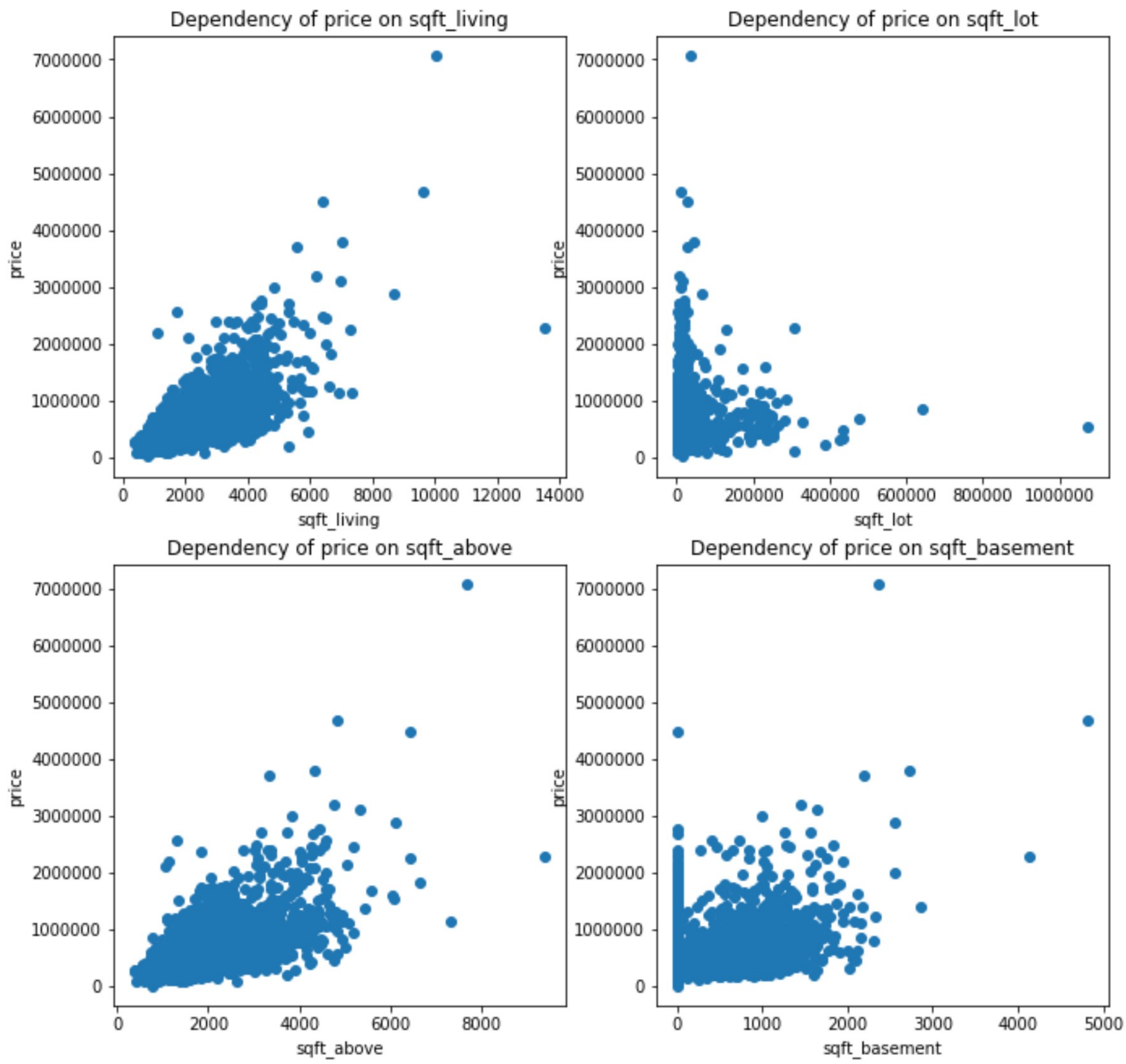
Out[253]:
```
Text(0, 0.5, 'price')
```



It seems that there is a positive correlation between price and each of the sqft variables. Let us remove the outliers to confirm this hypothsis.

In [254]:
```
df_outlier_removed = df[df["price"] < 1e7]

fig, ax = plt.subplots(2,2,figsize=(11,11))

ax[0,0].scatter(df_outlier_removed["sqft_living"],df_outlier_removed["price"])
ax[0,0].title.set_text("Dependency of price on sqft_living")
ax[0,0].set_xlabel("sqft_living")
ax[0,0].set_ylabel("price")

ax[0,1].scatter(df_outlier_removed["sqft_lot"],df_outlier_removed["price"])
ax[0,1].title.set_text("Dependency of price on sqft_lot")
ax[0,1].set_xlabel("sqft_lot")
ax[0,1].set_ylabel("price")

ax[1,0].scatter(df_outlier_removed["sqft_above"],df_outlier_removed["price"])
ax[1,0].title.set_text("Dependency of price on sqft_above")
ax[1,0].set_xlabel("sqft_above")
ax[1,0].set_ylabel("price")

ax[1,1].scatter(df_outlier_removed["sqft_basement"],df_outlier_removed["pric
e"])
ax[1,1].title.set_text("Dependency of price on sqft_basement")
ax[1,1].set_xlabel("sqft_basement")
ax[1,1].set_ylabel("price")
```

Out[254]: Text(0, 0.5, 'price')



Removing the outliers confirms the hypothesis that price is positively correlated with the four sqft features. In the cases of sqft_living and sqft_above, the correlation is strong, and it would be appropriate to model the data with Least Squares Linear Regression.

2.4 Train-test split and preprocessing pipeline

Analyzing the .dat file data reveals that the yr_renovated column has NaN values. In the .csv file, these NaN values have been replaced by zeros. We will set them back to np.nan and apply a simple median imputer to the yr_renovated column. We choose the median as the median of the years will always be an integer value. This will help us when we encode the yr_renovated column as a categorical variable.

In [255]:
```
df['yr_renovated'].mask(df['yr_renovated'] == 0, np.nan, inplace=True)
df['yr_renovated'].head()
```

Out[255]:
```
0    2005.0
1       NaN
2       NaN
3       NaN
4    1992.0
Name: yr_renovated, dtype: float64
```

In [256]:
```
#We preprocess the data without scaling
warnings.filterwarnings('ignore')

df = df_outlier_removed[df_outlier_removed.columns.difference(["price","sqft_li
ving","sqft_lot","sqft_above","sqft_basement"])].astype("object")
df = pd.concat([df,df_outlier_removed[["sqft_living","sqft_lot","sqft_above","s
qft_basement"]].astype("float")],axis=1)

categorical = df.dtypes == object

preprocess_unscaled = make_column_transformer((SimpleImputer(strategy="media
n"),["yr_renovated"]),(ce.OneHotEncoder(), categorical),("passthrough",~categor
ical))

X_train, X_test, y_train, y_test = train_test_split(df,df_outlier_removed["pric
e"])

model_lin_reg = make_pipeline(preprocess_unscaled, LinearRegression())
model_lasso = make_pipeline(preprocess_unscaled, Lasso())
model_ridge = make_pipeline(preprocess_unscaled,Ridge())
model_elastic_net = make_pipeline(preprocess_unscaled,ElasticNet())

scores_lin_reg = np.mean(cross_val_score(model_lin_reg, X_train, y_train))
scores_lasso = np.mean(cross_val_score(model_lasso,X_train,y_train))
scores_ridge = np.mean(cross_val_score(model_ridge,X_train,y_train))
scores_elastic_net = np.mean(cross_val_score(model_elastic_net,X_train,y_trai
n))

print("Linear Regression without scaling Score : "+str(scores_lin_reg))
print("Lasso without scaling Score : "+str(scores_lasso))
print("Ridge without scaling Score : "+str(scores_ridge))
print("Elastic Net without scaling Score : "+str(scores_elastic_net))
```

```
Linear Regression without scaling Score : -59.89561228919964
Lasso without scaling Score : 0.4687220016349805
Ridge without scaling Score : 0.7223574662665617
Elastic Net without scaling Score : 0.5463439805193165
```

In [257]:
```
#We preprocess the data with scaling
warnings.filterwarnings('ignore')

df = df_outlier_removed[df_outlier_removed.columns.difference(["price","sqft_li
ving","sqft_lot","sqft_above","sqft_basement"])].astype("object")
df = pd.concat([df,df_outlier_removed[["sqft_living","sqft_lot","sqft_above","s
qft_basement"]].astype("float")],axis=1)

categorical = df.dtypes == object

preprocess_scaled = make_column_transformer((SimpleImputer(strategy="media
n"),["yr_renovated"]),(ce.OneHotEncoder(), categorical),(StandardScaler(),~cate
gorical))

X_train, X_test, y_train, y_test = train_test_split(df,df_outlier_removed["pric
e"])

model_lin_reg = make_pipeline(preprocess_scaled, LinearRegression())
model_lasso = make_pipeline(preprocess_scaled, Lasso())
model_ridge = make_pipeline(preprocess_scaled,Ridge())
model_elastic_net = make_pipeline(preprocess_scaled,ElasticNet())

scores_lin_reg = np.mean(cross_val_score(model_lin_reg, X_train, y_train))
scores_lasso = np.mean(cross_val_score(model_lasso,X_train,y_train))
scores_ridge = np.mean(cross_val_score(model_ridge,X_train,y_train))
scores_elastic_net = np.mean(cross_val_score(model_elastic_net,X_train,y_trai
n))

print("Linear Regression with scaling Score : "+str(scores_lin_reg))
print("Lasso with scaling Score : "+str(scores_lasso))
print("Ridge with scaling Score : "+str(scores_ridge))
print("Elastic Net with scaling Score : "+str(scores_elastic_net))
```

```
Linear Regression with scaling Score : -7917.8215704149015
Lasso with scaling Score : 0.3756445427070364
Ridge with scaling Score : 0.7363816409099707
Elastic Net with scaling Score : 0.5306898757486109
```

Scaling does not improve results, so we will not use scaling going forward.

1.5 Tuning the parameters

In [258]:
```
#We Dummy encode the categorical variables, impute yr_renovated and train-test split

df = df_outlier_removed[df_outlier_removed.columns.difference(["price","sqft_living","sqft_lot","sqft_above","sqft_basement"])].astype("object")
df = pd.concat([df,df_outlier_removed[["sqft_living","sqft_lot","sqft_above","sqft_basement"]].astype("float")],axis=1)

categorical = df.dtypes == object

preprocess_scaled = make_column_transformer((SimpleImputer(strategy="median"),["yr_renovated"]),(ce.OneHotEncoder(), categorical),("passthrough",~categorical))

df = make_pipeline(preprocess_scaled).fit_transform(df)

X_train,X_test,y_train,y_test = train_test_split(df,df_outlier_removed["price"])
X_train_2,X_validate,y_train_2,y_validate = train_test_split(X_train,y_train)

print("After dummy encoding and median imputing, X_train has the following shape : "+str(np.shape(X_train)))
```

```
After dummy encoding and median imputing, X_train has the following shape : (3411, 4828)
```

Dummy encoding all the categorical variables has created $\approx$ 4800 more features. This explains why our Linear Regression Model performs so badly. For OLS, we need to have more rows than we have features.

In [259]:
```
#warnings.filterwarnings('ignore')

pipe = Pipeline([("regressor",LinearRegression())])

param_grid = [{'regressor': [LinearRegression()]},
              {'regressor': [Lasso()],
               'regressor__alpha':np.logspace(-3,3,10)},
              {'regressor': [Ridge()],
              'regressor__alpha':np.logspace(-3,3,10)},
              {'regressor': [ElasticNet()],
              'regressor__alpha':np.logspace(-3,3,10),
              'regressor__l1_ratio':np.logspace(-3,0,10)}]

grid = GridSearchCV(pipe, param_grid,cv=5)
grid.fit(X_train_2,y_train_2)

print("Best model based on training : "+ str(grid.best_params_))
print("Score of best model on test : "+str(grid.score(X_validate,y_validate)))
```

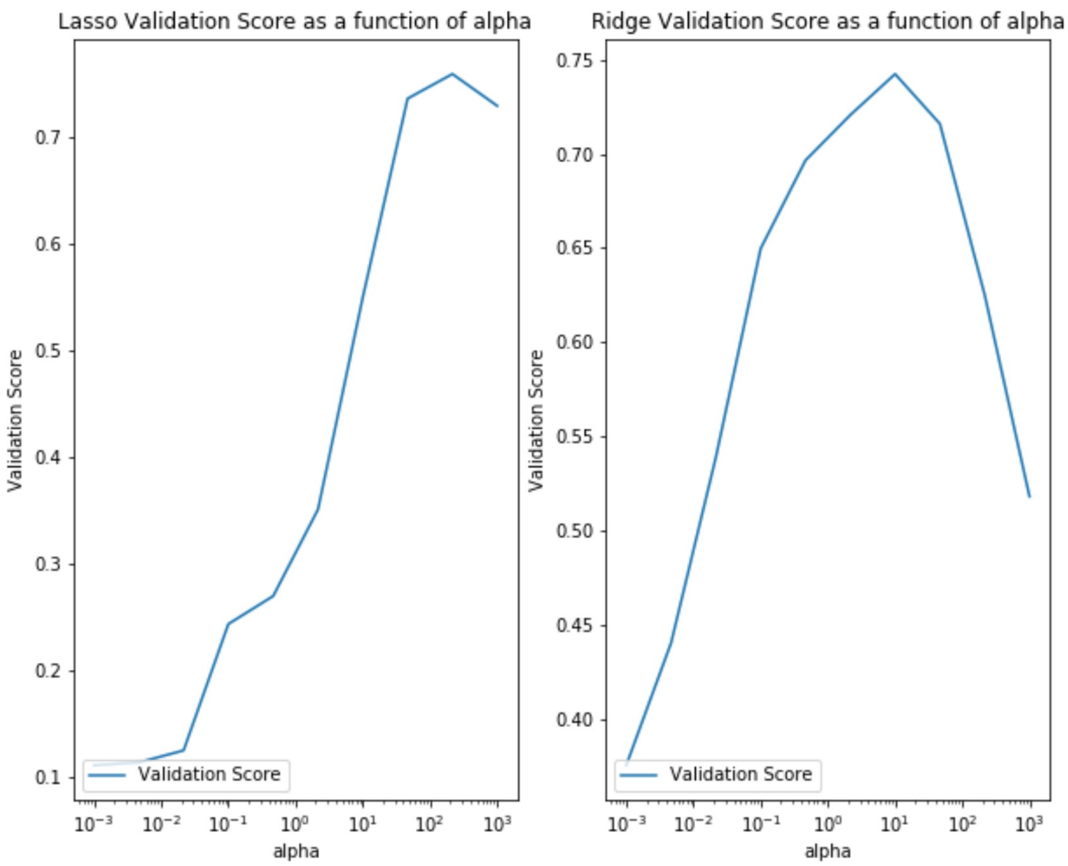
```
Best model based on training : {'regressor': Lasso(alpha=215.44346900318823, copy_X=True, fit_intercept=True, max_iter=1000,
      normalize=False, positive=False, precompute=False, random_state=None,
      selection='cyclic', tol=0.0001, warm_start=False), 'regressor__alpha': 215.44346900318823}
Score of best model on test : 0.7583617978140507
```

Parameter Tuning improves results of the best regressor. The best model is a Lasso with alpha=215. Let us now see how the validation scores depend on the parameters

In [260]:
```
#We compute the dependence of validation scores on parameters for Lasso and Rid
ge
Lasso_params = np.logspace(-3,3,10)
Lasso_vals = [Lasso(alpha=Lasso_params[i]).fit(X_train_2,y_train_2).score(X_val
idate,y_validate) for i in range(len(Lasso_params))]

Ridge_params = np.logspace(-3,3,10)
Ridge_vals = [Ridge(alpha=Ridge_params[i]).fit(X_train_2,y_train_2).score(X_val
idate,y_validate) for i in range(len(Lasso_params))]
```

In [261]:
```
fig, ax = plt.subplots(1,2,figsize=(10,8))

ax[0].plot(Lasso_params,Lasso_vals,label="Validation Score")
ax[0].set_xlabel("alpha")
ax[0].set_xscale("log")
ax[0].set_ylabel("Validation Score")
ax[0].title.set_text("Lasso Validation Score as a function of alpha")
ax[0].legend(loc="lower left")

ax[1].plot(Ridge_params,Ridge_vals,label="Validation Score")
ax[1].set_xlabel("alpha")
ax[1].set_xscale("log")
ax[1].set_ylabel("Validation Score")
ax[1].title.set_text("Ridge Validation Score as a function of alpha")
ax[1].legend(loc="lower left")
```

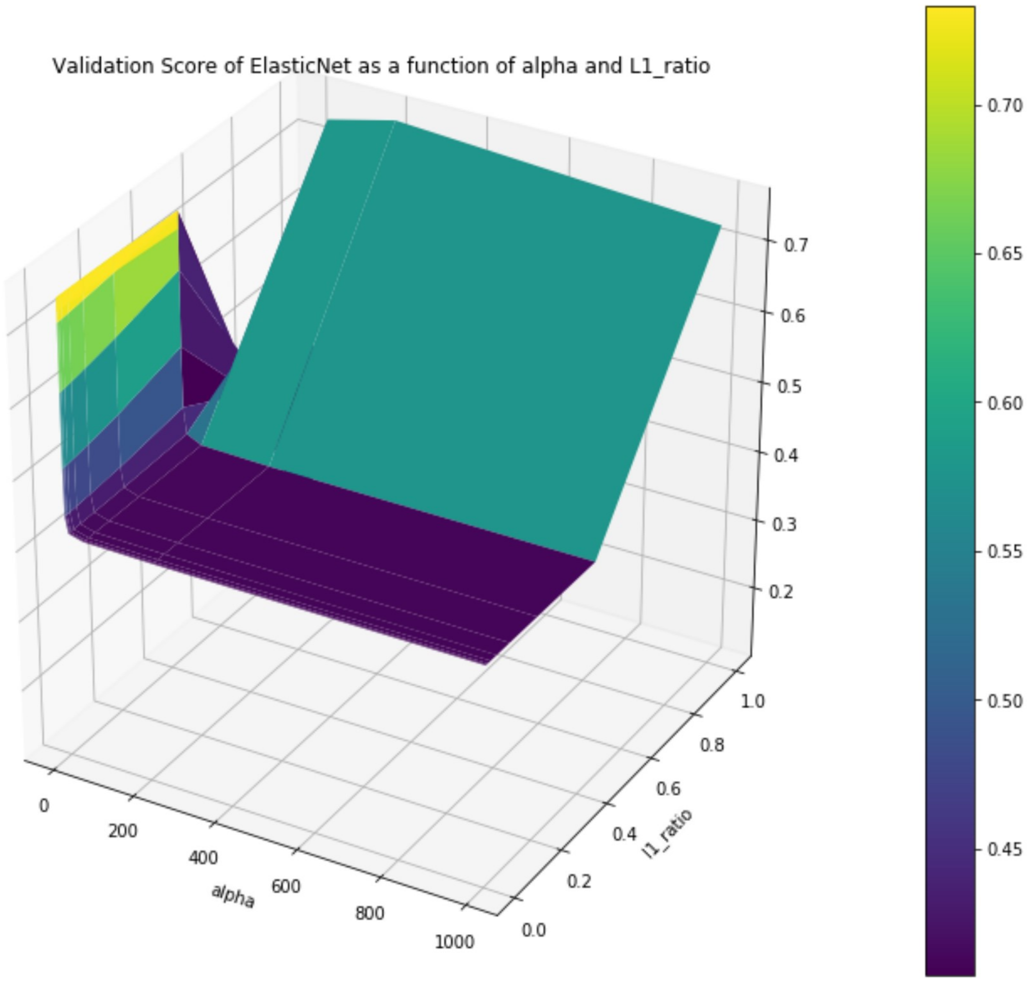
Out[261]: <matplotlib.legend.Legend at 0x11bc5802cc0>



In [267]:
```
#We compute the dependence of validation scores on parameters for ElasticNet

from mpl_toolkits.mplot3d import Axes3D

x = np.logspace(-3,3,10)
y = np.logspace(-3,0,10)
xx, yy = np.meshgrid(x, y, sparse=False, indexing='xy')
zz = np.zeros((10,10))

for i in range(10):
    for j in range(10):
        zz[i,j] = ElasticNet(alpha=xx[i,j],l1_ratio=yy[i,j]).fit(X_train_2,y_tr
ain_2).score(X_validate,y_validate)
```

In [273]:
```
#We plot the dependence of validation scores
%matplotlib inline

fig = plt.figure(figsize=(10,8))
ax = Axes3D(fig)

surf = ax.plot_surface(xx,yy,zz,cmap='viridis')
plt.ylabel('l1_ratio')
plt.xlabel('alpha')
plt.title('Validation Score of ElasticNet as a function of alpha and L1_ratio')
fig.colorbar(surf)
```

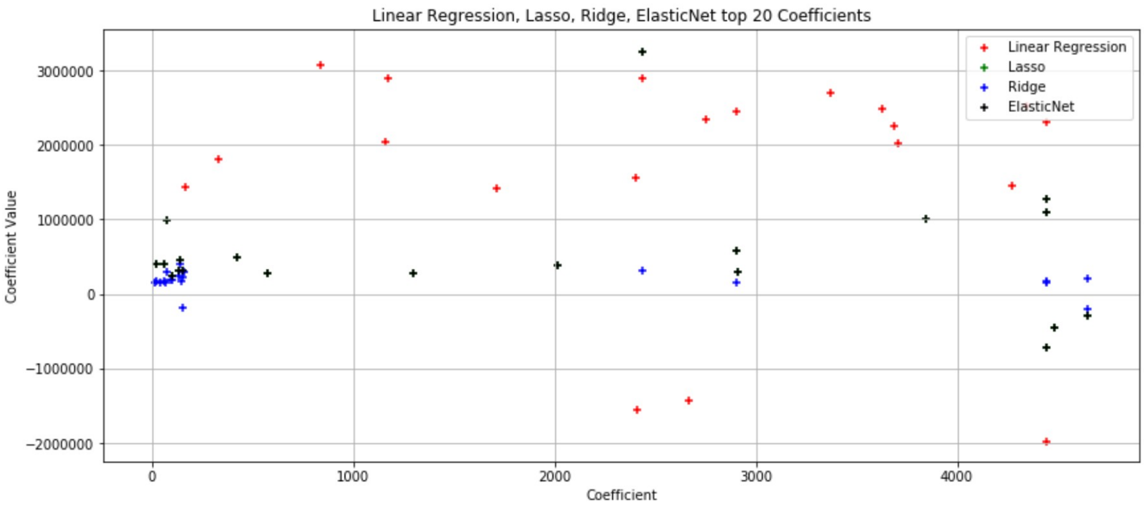
Out[273]: `<matplotlib.colorbar.Colorbar at 0x11b9dac73c8>`



2.6 20 Most important coefficients of each model

In [264]:
```
#We first select satisfactory parameters for each model

#Linear Regression
model_LR = LinearRegression().fit(X_train,y_train)

#Lasso Regression
pipe = Pipeline([('regressor',Lasso())])

param_grid = [{'regressor': [Lasso()],
               'regressor__alpha':np.logspace(-3,3,10)}]

grid = GridSearchCV(pipe, param_grid)
grid.fit(X_train,y_train)

model_Lasso = Lasso(alpha = grid.best_params_["regressor__alpha"]).fit(X_train,
y_train)

#Ridge Regression
pipe = Pipeline([('regressor',Ridge())])

param_grid = [{'regressor': [Ridge()],
               'regressor__alpha':np.logspace(-3,3,10)}]

grid = GridSearchCV(pipe, param_grid)
grid.fit(X_train,y_train)

model_Ridge = Ridge(alpha = grid.best_params_["regressor__alpha"]).fit(X_train,
y_train)

#ElasticNet Regression
pipe = Pipeline([('regressor',ElasticNet())])

param_grid = [{'regressor': [ElasticNet()],
               'regressor__alpha':np.logspace(-3,3,10),
               'regressor__l1_ratio':np.logspace(-3,0,10)}]

grid = GridSearchCV(pipe, param_grid)
grid.fit(X_train,y_train)

model_Elastic_Net = ElasticNet(alpha = grid.best_params_["regressor__alpha"],l1
_ratio=grid.best_params_["regressor__l1_ratio"]).fit(X_train,y_train)
```

In [271]:
```
%matplotlib inline
#We select the 20 most important coefficients (i.e those with largest absolute value) for Logistic Regression
coef_LR = np.absolute(model_LR.coef_)
coef_LR = np.argsort(coef_LR) #sorts from smallest to largest
coef_LR = np.flip(coef_LR)[:20] #selects 20 most important coefs

coef_Lasso = np.absolute(model_Lasso.coef_)
coef_Lasso = np.argsort(coef_Lasso)
coef_Lasso = np.flip(coef_Lasso)[:20]

coef_Ridge = np.absolute(model_Ridge.coef_)
coef_Ridge = np.argsort(coef_Ridge)
coef_Ridge = np.flip(coef_Ridge)[:20]

coef_Elastic_Net = np.absolute(model_Elastic_Net.coef_)
coef_Elastic_Net = np.argsort(coef_Elastic_Net)
coef_Elastic_Net = np.flip(coef_Elastic_Net)[:20]


plt.figure(figsize=(14,6))

plt.xlabel("Coefficient")
plt.ylabel("Coefficient Value")
plt.grid()

plt.scatter(coef_LR,[model_LR.coef_[i] for i in coef_LR],marker="+",c="r",label="Linear Regression")
plt.scatter(coef_Lasso,[model_Lasso.coef_[i] for i in coef_Lasso],marker="+",c="g",label="Lasso")
plt.scatter(coef_Ridge,[model_Ridge.coef_[i] for i in coef_Ridge],marker="+",c="b",label="Ridge")
plt.scatter(coef_Elastic_Net,[model_Elastic_Net.coef_[i] for i in coef_Elastic_Net],marker="+",c="k",label="ElasticNet")
plt.title("Linear Regression, Lasso, Ridge, ElasticNet top 20 Coefficients")
plt.legend(loc="upper right")

plt.show()
```



The order of magnitude of the coefficients is 10^6, which is the same order of magnitude that price has. This indicates that our models have not overfitted.

The plot indicates that some of the coefficients are aligned on the same x (coefficient) values. This means that the models have selected the same features. Let us confirm this by looking directly at the top 20 coefficients.

In [272]:
```
print("Linear Regression Coefs : "+str(np.sort(coef_LR)))
print("Lasso Coefs : "+str(np.sort(coef_Lasso)))
print("Ridge Coefs : "+str(np.sort(coef_Ridge)))
print("Elastic Net Coefs : "+str(np.sort(coef_Elastic_Net)))
```

```
Linear Regression Coefs : [ 170  331  836 1161 1173 1713 2407 2414 2439 2670 2
753 2902 3370 3627
 3689 3706 4276 4343 4446 4447]
Lasso Coefs : [  20   64   74  101  135  139  153  421  573 1296 2018 2439 290
2 2913
 3847 4445 4446 4447 4481 4648]
Ridge Coefs : [  16   20   40   64   67   74   93  101  135  139  149  153  15
7  163
 2439 2902 4445 4446 4648 4649]
Elastic Net Coefs : [  20   64   74  101  135  139  153  421  573 1296 2018 24
39 2902 2913
 3847 4445 4446 4447 4481 4648]
```

The top 20 coefficients in all 4 models are similar. The top 20 coefficients for Lasso, Ridge and ElasticNet are almost identical.

In [ ]: