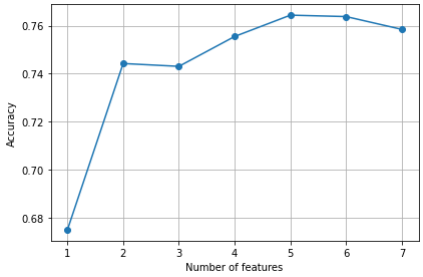
Model 1: Random Forest

Business Problem:
A business or individual would certainly want to predict how popular an app will be. One reasonable way to define that is based on the number of installs. We can consider an app with over 1,000,000 downloads to be popular and those with downloads below that to be unpopular. We want to build a model to predict whether a new app will be a popular app.

Analysis and Modeling:
After cleaning the data, there were 8892 rows. 2566 apps had over 1,000,000 install or more, and 6326 apps had installs less than that. I set aside 20% of the data for testing allocating around the same proportion of over and less than 1,000,000 install apps as the training set.

I did not apply standardization on the data for the random forest. Using the backward selection algorithm (from one of the homeworks), I decided to go with 5 features. However, deleting variables is not extremely necessary for tree algorithms.



Random Forests are one of the most popular machine learning algorithms with good accuracy, robust to outliers, and less risk of overfitting. I used the data with the attributes Category, Rating, Price, Content Rating, Genres, and built a random forest model with the following parameters.

```
forest = RandomForestClassifier(n_estimators=500, max_features=3, max_depth=30, random_state=2, n_jobs=2)
forest.fit(X_train, y_train)
```

```
RandomForestClassifier(max_depth=30, max_features=3, n_estimators=500, n_jobs=2,
                       random_state=2)
```

```
forest.score(X_test, y_test)
```

```
0.7448194197750148
```

Using the data with the features Category, Rating, Price, Content Rating, Genres, the random forest model achieved close to a 75% accuracy, which is reasonable. Including the number of reviews in the model will dramatically increase the accuracy rate, however the number of reviews may not be known beforehand so I did not include it. Also, the confusion matrix can better illustrate the fit of the model.

```
y_pred_test = forest.predict(X_test)
confusion_matrix(y_test, y_pred_test)
```

```
array([[985, 137],
       [294, 273]], dtype=int64)
```

Model 2: K-Nearest Neighbor Regression

Business Problem:
An app developer may be interested in predicting the rating of the app. Better ratings will better reflect the app as well as the reputation of the developer or company. Apps with higher ratings could potentially attract more users. So it is importantly accurately predict the rating of an app. This is a regression problem.

Analysis and Modeling:
I applied a KNN algorithm to predicting the app ratings. I standardized the data as KNN is sensitive to scaling. I decided to add all the features to the model. KNN only has 1 parameter that needs tuning, which is the K value. A guideline online is to use k to be around squareroot of N. After experimenting, K=60 worked well for this data set. A more accurate K can be determined from cross-validation, which I can explore in the next part of the project (testing).

```
model2 = KNeighborsRegressor(n_neighbors=60, metric="euclidean")
model2.fit(X_train_std, y_train)
model2.score(X_test_std,y_test)
```

```
0.9260218114149913
```

We got a model with a R squared of 92.6%, which is good. Which means 92.6% of the variance in Rating is explained by the predictors. We can also get the MSE and RMSE of the model. And both numbers are fairly low.

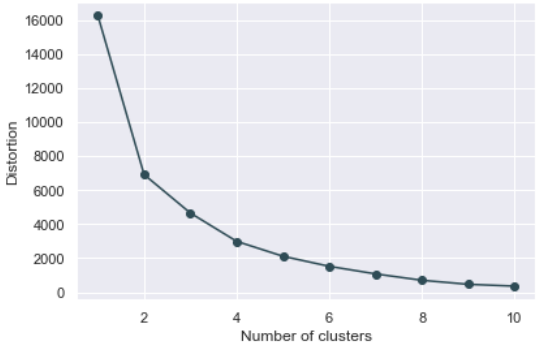
```
mse =mean_squared_error(y_test, model2.predict(X_test_std))
print("Mean Squared Error:",mse)
rmse = math.sqrt(mse)
print("Root Mean Squared Error:", rmse)
```

```
Mean Squared Error: 0.19302928966789668
Root Mean Squared Error: 0.439350986874841
```

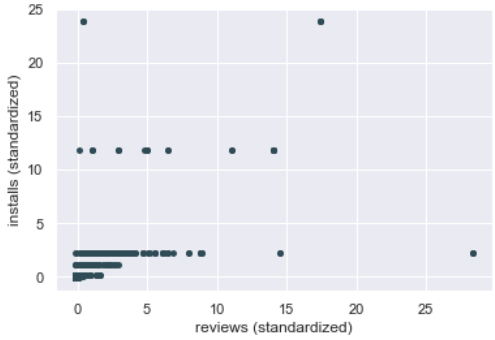
Model 3: K-Means Clustering (Unsupervised)

Business Problem:
A business might be interested in identifying apps where people have strong feelings about or apps that people don't have much to say about. In our scenario, we would define that as apps with high number of reviews but high number of downloads or apps with high number of downloads but low number of reviews. Apps with high number of reviews, but low downloads could also signal abnormalities, such as a lot of people posting review about a technical problem with the app. Information like these will be helpful for companies. So we are looking for clusters.

Analysis and Modeling:
K-means is an unsupervised algorithm that can identify clusters in a data set. I applied this algorithm to our data set with Reviews and Installs as features. For K-means it's important the features be the same scale. I standardized the two features using the standard scaler. We can pick K, the number of clusters, using the elbow method. From this graph, I decided to pick K=5.

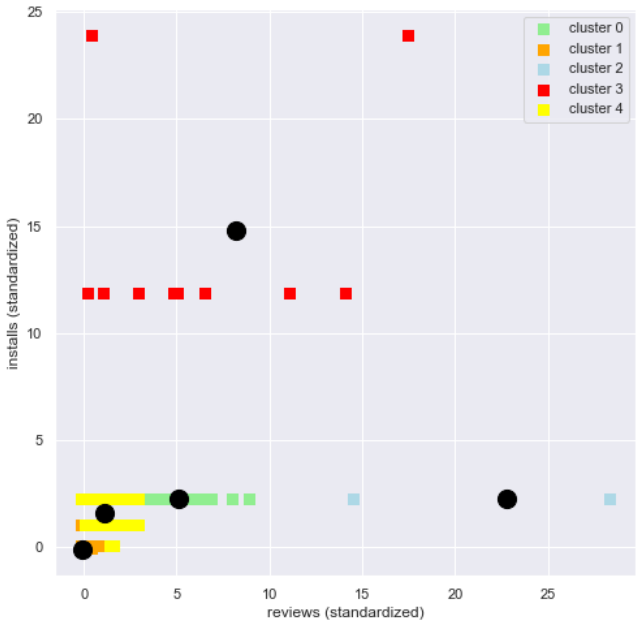


The initial plot of reviews and installs after standardizing looks like.




```
from sklearn.cluster import KMeans

km = KMeans(n_clusters=5,
            init='random',
            n_init=10,
            max_iter=300,
            tol=1e-04,
            random_state=0)
```



We can conclude that K=5 works well for the data. The black points are the cluster centroids. The red cluster are those apps with higher number of installs compared to reviews. The blue cluster are those apps with higher number of reviews compared to installs. One of the weaknesses of K-means is it's sensitive to outliers. I did not remove any outlier points for this scenario.