Advert Analysis Research

Part 1

How does feature selection affects overfitting and validation loss?

Andrey Verbovskiy

01.10.2021

Background and pre-analysis:

Not a long time ago I got a big dataset of a big well-known technology company and decided to experiment on it later. The dataset is dedicated to mobile users' data that received a mobile advert, the dataset also includes the info if the user installed the advert or not. Working on this dataset is interesting to me for two main reasons, first one is that so far I have not worked with such big number of samples (around 1 million), second reason is that this is a real working data that is hard to analyze and make a clear outcome out of it, not a prepared data for some assignments that is designed to make a clear and perfect outcome. This data set is quite complicated and has many features that made my research hard, however exactly for those kinds of challenges I decided to do this research with this dataset.

After a quick look on the data, I noticed that there are quite many features, so I asked myself if all the features are necessary for estimating the user's desire to install the app. This question made me think of what method would be the best in this case to select the most important features.

Goal:

The goal in this first stage of a big research is to analyze the best feature selection algorithm for big data that consists of both numerical and categorical values for future stages of research. This research also includes data preprocessing and building a neural network to test the effectiveness of the chosen feature selection algorithms.

Why is it important?

Feature selection is important factor in data preprocessing. Proper feature selection helps model in not learning useless features and trends, it reduces overfitting and increases computation speed as there is less data for model to handle. This research will investigate which method works best with mix of categorical and numerical inputs, as there are still many arguments on such issue.

Hypothesizes:

Hypothesize 1: As output is categorical and most of the input is categorical as well, in theory Filter methods such as chi-squared will work the best.

Hypothesize 2: Embedded methods such as Embedded random forest are known to have less troubles with mixed data, so in this case, this method will show the best performance.

Brief Data Analysis:

In the provided dataframe there was a lot of interesting data. In this section I briefly explain my thoughts concerning each data column and what operations were required to be done with the data.

1) Impression id:

As impression id is randomly generated and is unique for each data sample, there is no real need in it for the analysis and creation of model. During preprocessing this column will be dropped as it can be replaced by default index column.

2)Timestamp:

The significance of timestamp is questionable, as how would time and date of the event affect the motivation of user to install the add. On one hand, there is no direct pattern, on the other hand, if we assume that it was a weekend, when people are more likely to install some party games, or some other pattern that justifies the presence of this data.

3) campaignid:

This column is one of the most essential ones, there are quite many unique campaigns that directly influence the chances of installation.

4) platform:

This criteria is relatively important as it specifies what platform the user has. Platform gives us the information about the application store that user has access to. It is important as some apps might not be available on every store. This data is not one of the most important ones, yet carries some meaning that can be useful for analysis.

5) softwareVersion

Just like the previous data, this one is useful to understand the users device better. Sometimes, there can be mobile applications, that cannot be installed on old software, what makes it useless to advertise it to people with old software devices.

6) sourceGameId:

This feature shows what app/game the user was using when the advertisement appeared. This data sometimes can be extremely significant, because if a user plays an action game, sending advertisement of a chess game would most likely be useless, unless the user would like to try something new. However, mostly the user will just skip the ad, but if this user gets an ad of a game with the same genre or other parameter, his installation probability must increase. So, considering the game that the user plays at the moment can be crucial.

7) country:

In most of the cases this column is useless as there is no real relation of it to installation chance. The only exceptions are games, that are deliberately designed for a certain country or region, this can be represented in a way of specific humor or connected to other country’s feature.

8) startCount:

This is one of the most significant data parts. This data describes the activity of the user.

9) viewCount:

This feature shows how many adds the user has seen. This data is also one of the most important ones. It illustrates the interest of the person in the ads.

10) clickCount:

This feature shows how many adds the user has clicked on. Slightly more important data than the previous one, illustrates the interest as well. Yet, this one user not only saw the advert, but also clicked on it to obtain more information. However, even if the user clicked on it, it does not mean that the app was installed.

11) installCount:

This feature shows how many adds the user has installed before. This is most likely the most important data in this dataframe. It actually shows how actively user installs the apps. Based on this criteria, it can be defined how easy it is to make this user install the app. Some users can easily install apps because of their curiosity, some don't install at all despite the number of times they clicked on adds.

12) lastStart:

This data can also be useful as it shows the last time user started any advertisement campaign. If the last session was a long time ago, the user either stopped playing the game or uses add block or related program. In both cases the ad should not be wasted on this case.

13) startCount1d:

This data serves the same purpose as the previous one, yet it holds a more specified data – the last 24 hours. Based on this data, a daily activity of the user can be approximately estimated.

14) startCount7d:

Carries the same mission as the previous data. However, in my opinion this one is more significant as, for example, if the user has been actively starting campaigns for 6 days and could not do that on the last day, startCount7d will give him a high score, yet startCount1d would give him lower one and that would be a mistake.

15) connectionType:

This data is relevant in case if a user has limited internet connection, which does not allow the user to install the app right away and forces to install it later, when there is a more stable internet connection.

Preparing the Data:

In the training data preprocessing I started with reading the csv file and getting the descriptive statistics.

```
data.describe().round(2)
```

	sourceGameId	startCount	viewCount	clickCount	installCount	startCount1d	startCount7d	install
count	1.000001e+06	1000001.00	1000001.00	1000001.00	1000001.00	1000001.00	1000001.0	1000001.00
mean	3.114280e+06	46.14	34.26	2.04	0.54	6.47	18.8	0.01
std	1.397085e+07	87.89	76.56	6.71	1.51	11.13	34.0	0.12
min	1.105500e+04	1.00	0.00	0.00	0.00	1.00	1.0	0.00
25%	1.196472e+06	5.00	2.00	0.00	0.00	1.00	3.0	0.00
50%	1.623384e+06	17.00	9.00	0.00	0.00	3.00	8.0	0.00
75%	2.633648e+06	50.00	32.00	2.00	0.00	7.00	21.0	0.00
max	1.316274e+08	3997.00	2886.00	1031.00	73.00	1202.00	3280.0	1.00

Figure 1: Descriptive Statistics.

Then, I was exploring data by calling unique() function for all the columns (I deleted most of them to keep code cleaner), listing the number of missing values in each column and listing the types of data in columns.

```
#looking for missing values
data.isna().sum()
```

```
id                    0
timestamp             0
campaignId            0
platform              0
softwareVersion       0
sourceGameId          0
country              39
startCount            0
viewCount             0
clickCount            0
installCount          0
lastStart         78824
startCount1d          0
startCount7d          0
connectionType        0
install               0
dtype: int64
```

Figure 2: unique function and types of data in columns.

Most of the NaN values were present in lastStart column and just a few in country (probably because of VPN alike apps). The Nan values were dropped as in this first stage we just test the feature selection methods, we do not aim to make a perfect model yet. To simplify the research the samples with missing values got deleted.

Then I checked what data types are present in the dataset, as for further calculations we require integer values:

```
#check the data types
data.dtypes
```

```
id                 object
timestamp          object
campaignId         object
platform           object
softwareVersion    object
sourceGameId        int64
country            object
startCount          int64
viewCount           int64
clickCount          int64
installCount        int64
lastStart          object
startCount1d        int64
startCount7d        int64
connectionType     object
install             int64
dtype: object
```

Figure 3: Object types.

Also, I checked how many app installations are present in the dataset:

```
data['install'].value_counts()
```

```
0    908976
1     12167
Name: install, dtype: int64
```

Figure 3: Number of app installations.

As it can be seen, most of the users did not install the advert, which is not surprising. However, it will make next stages of research much more complicated as it will be hard for a neural network to detect patterns leading to installation and just giving 0 as an outcome will give high accuracy. Yet this challenge is relevant only for next stage of research.

Next step was to convert all the object types into integer values:

```
softwareVersion_unique_values = data['softwareVersion'].unique()

data['softwareVersion'].replace(softwareVersion_unique_values, np.arange(0,len(softwareVersion_unique_values),1), inplace=True)

sourceGameId_unique_values = data['sourceGameId'].unique()

country_unique_values = data['country'].unique()

data['country'].replace(country_unique_values, np.arange(0,len(country_unique_values),1), inplace=True)
```

Figure 4: Example of values replacement.

The most time-consuming replacement was that of campaignId as there were around 2327 unique values. It took quite some time for the replacement process to be completed.

The most complicated thing in this part was converting time columns. I did spend some time figuring out how should I efficiently convert it, I was mainly trying to convert it myself by some calculations. At some point I discovered that importing time and datetime can help me. After that I quickly created a function called time_convert, which was designed to return numerical timestamps.

```
In [22]: def time_convert(x):
             try:
                 return pd.Timestamp(x, tz='utc').timestamp()
             except:
                 return x
```

Figure 5: time_convert function.

Final step was to drop irrelevant column (id) and confirm that there are no NaN values left.

Also, to have a quick look at data correlations, I plotted a correlation matrix:

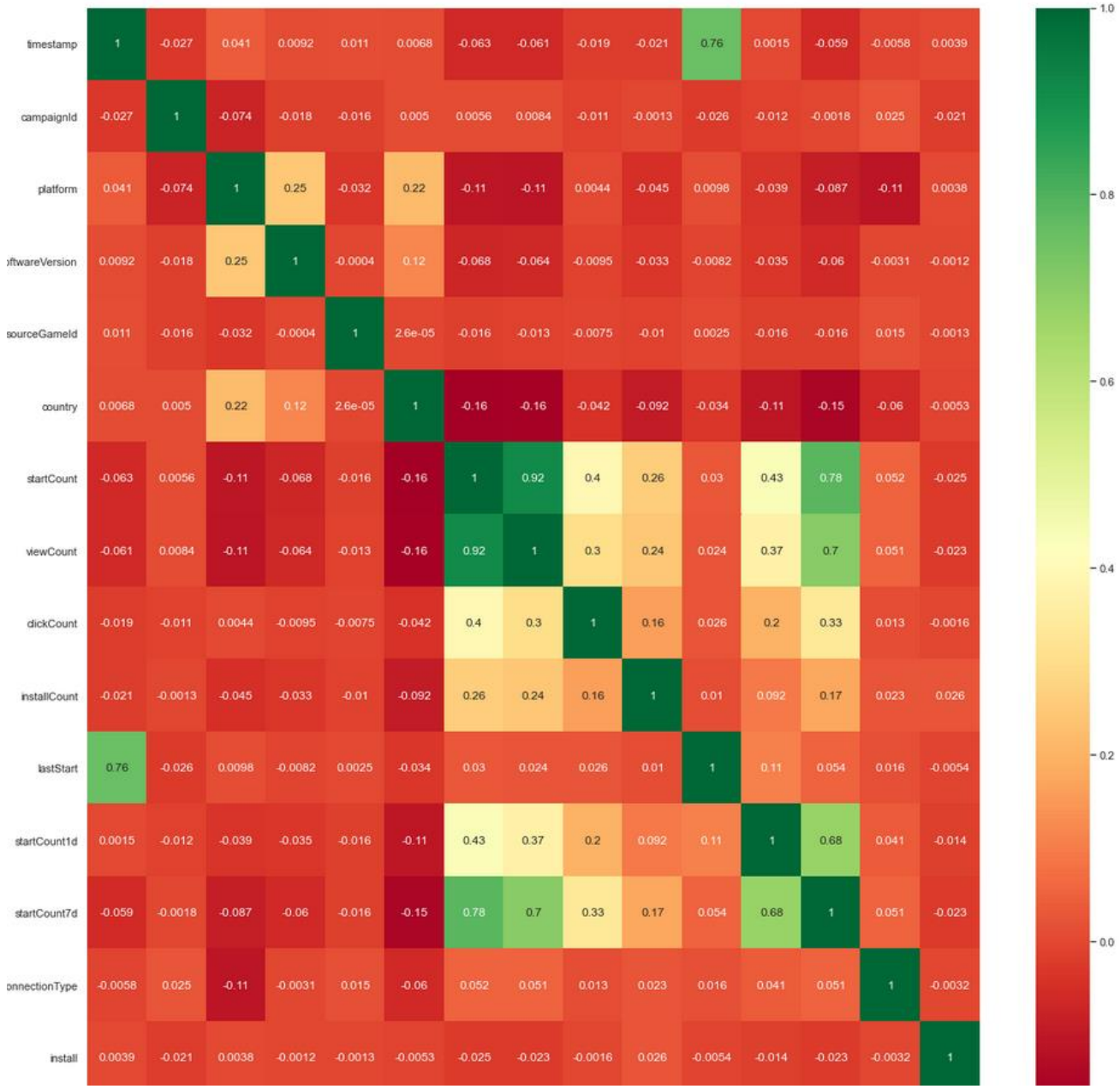


Figure 6: Correlation matrix.

Data split:

Installation column goes to Y values and the rest features go to X set. To make future analysis easier and more efficient for the model, the X values should be normalized:

```
#preparing data
X = np.array(data.drop(['install'], 1))
y = np.array(data['install'])
```

```
from sklearn.preprocessing import MinMaxScaler
X_norm = MinMaxScaler().fit_transform(X)
```

Figure 7: X and Y data split. Normalization of X values

Next step was to split dataset into training and validation/testing dataset. The second one will later be splitted into separate validation and testing datasets. Randomly chosen, 80% of all the data will go into training sets as we want model to be properly trained. In this stage of research, we will not need the testing dataset that much, but we still will prepare it for future.

```
X_train, X_val_and_test, Y_train, Y_val_and_test = train_test_split(X_norm, y, test_size=0.2, random_state=1)

print(X_train.shape, X_val_and_test.shape, Y_train.shape, Y_val_and_test.shape)
```

```
(736914, 14) (184229, 14) (736914,) (184229,)
```

Figure 8: Train and validation/test data splits.

Feature selection methods:

In order to answer the research question, I need to select various feature selection methods. There are three main categories of feature selection methods, depending on their interaction with classification problems: Filter, Wrapped and Embedded. Filter methods rank features depending on parameters such as variance and correlation. The wrapper method uses machine learning algorithm as a tool to estimate feature importance. Embedded method is a mix of previous methods it ranks feature importance and then uses ML algorithm to estimate classification performance of each subset.

In order to test filter methods, I chose chi-square and mutual information methods. The chi-square method calculates divergence from expected distribution of a class value. It checks relationship between feature and target value, then gives the feature a score of importance.

Firstly, we use SelectKbest model and fit our normalized X and Y in it. After that we transform train and test datasets for future use, according to feature importance.

To have a clear understanding of what features are important according to chi-squared method, the histogram can be plotted. As it can be seen, not that many features received a high score, the most important ones are campaignId, startCount, viewCount and installCount, which seem logical:

```
from sklearn.feature_selection import SelectKBest
from sklearn.feature_selection import chi2

chi = SelectKBest(chi2, k='all')
chi.fit(X_norm, y)

#prepare x and y data for future modelling
X_train_chi = chi.transform(X_train)
X_test_chi = chi.transform(X_val_and_test)
```

```
for i in range(len(chi.scores_)):
    print('Feature %d: %f' % (i, chi.scores_[i]))
# plot of the features importance
plt.bar([i for i in range(len(chi.scores_))], chi.scores_)
plt.show()
```

```
Feature 0: 2.635800
Feature 1: 67.194926
Feature 2: 3.398083
Feature 3: 0.057697
Feature 4: 0.688321
Feature 5: 3.483883
Feature 6: 24.267319
Feature 7: 27.458421
Feature 8: 0.050409
Feature 9: 35.304128
Feature 10: 0.492544
Feature 11: 3.415438
Feature 12: 9.738913
Feature 13: 1.567649
```

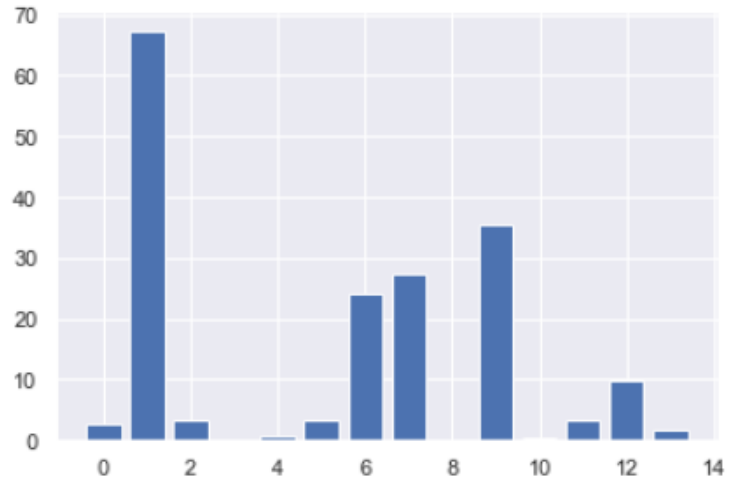


Figure 7: Chi-square feature selection method and it's result.

Next filter method is called mutual information, it estimates every features importance separately by estimating its decrease in entropy. The process of getting the feature selection is similar to that of chi-squared, we create a model based on mutual information classification and fit normalized X and Y values. Transform train and test data for suture and plot imporance of all features, as it can be seen, in mutual information's opinion only two factors are relevant, platform and startCount7d, which is quite surprising result:

Mutual information

```
from sklearn.feature_selection import mutual_info_classif

mut_inf = SelectKBest(mutual_info_classif, k='all')
mut_inf.fit(X_norm, y)

#prepare x and y data for future modelling
X_train_mut = mut_inf.transform(X_train)
X_test_mut = mut_inf.transform(X_val_and_test)
```

```
for i in range(len(mut_inf.scores_)):
    print('Feature %d: %f' % (i, mut_inf.scores_[i]))
# plot of the features importance
plt.bar([i for i in range(len(mut_inf.scores_))], mut_inf.scores_)
plt.show()
```

```
Feature 0: 0.000061
Feature 1: 0.005160
Feature 2: 0.117731
Feature 3: 0.000973
Feature 4: 0.001865
Feature 5: 0.001346
Feature 6: 0.000798
Feature 7: 0.000496
Feature 8: 0.000211
Feature 9: 0.000309
Feature 10: 0.000162
Feature 11: 0.000285
Feature 12: 0.000651
Feature 13: 0.051020
```

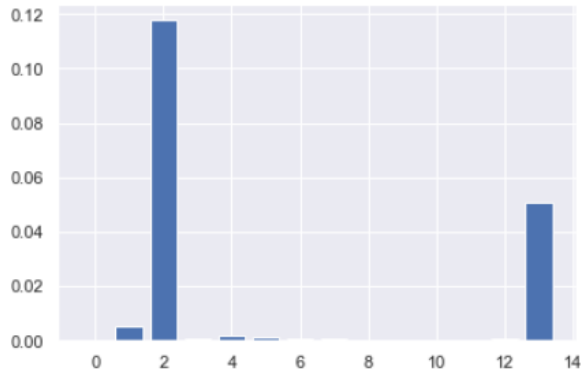


Figure 8: Mutual information feature importance.

Next step is implementation of Wrapper method, which is Recursive Feature Elimination. It puts data in ExtraTreesClassifier model, then based on model's performance analyzes how much does each feature

contributes to the target variable. To obtain the feature importance the inbuild function 'feature_importance' can be used. This method does not support automatic data transformation, so all unnecessary features had to be dropped by hand. According to this method, the most irrelevant features are platform, connectionType and installCount. After dropping the values, I normalized and splited dataset into train and validation/test datasets:

Wrapper Method

Recursive Feature Elimination

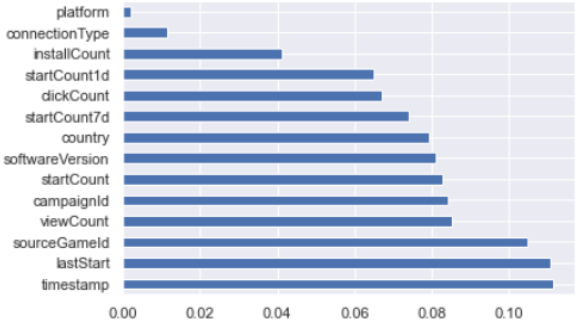
```
from sklearn.ensemble import ExtraTreesClassifier

rfe = ExtraTreesClassifier()
rfe.fit(X_norm,y)
```

```
ExtraTreesClassifier()
```

```
print(rfe.feature_importances_)
```

```
[0.11140641 0.08413764 0.00214745 0.08098326 0.10481663 0.07925356
 0.08281129 0.08504993 0.06696052 0.04131481 0.11073221 0.06488631
 0.07399935 0.01150063]
```

```
# Top important feeatures
imp = pd.Series(rfe.feature_importances_, index=data.iloc[:,0:14] .columns)
imp.nlargest(14).plot(kind='barh')
plt.show()
```




```
#dropping the useless data
X2 = np.array(data.drop(['install','platform','connectionType','installCount'], 1))
```

```
X2_norm = MinMaxScaler().fit_transform(X2)
```

```
X2_train, X2_val_and_test, Y2_train, Y2_val_and_test = train_test_split(X2_norm, y, test_size=0.2, random_state=1)

print(X2 train.shape, X2 val and test.shape, Y2 train.shape, Y2 val and test.shape)
```

Figure 9: Recursive Feature Elimination method.

Finally, as Embedded method the Embedded Random Forest was chosen. This method randomly permutes feature and calculates the misclassification rate compared to rate of all features performance. In oreder to perform such thing it uses RandomForestClassifier. To see what features are relevant, the '.get_support()' method can be used. It represents relevant features as True value, in this case only 4 features have significance: timestamp, campainId, sourceGameId and lastStart:

Embedded Method

Embedded Random Forest

```
from sklearn.feature_selection import SelectFromModel
from sklearn.ensemble import RandomForestClassifier

erf = SelectFromModel(RandomForestClassifier(n_estimators=100))
erf.fit(X_norm, y)
```

```
SelectFromModel(estimator=RandomForestClassifier())
```

```
#This array shows waht features are important(True) and not important(False)
erf.get_support()
```

```
array([ True,  True, False, False,  True, False, False, False, False,
       False,  True, False, False, False])
```

```
#prepare x and y data for future modelling
X_train_erf = erf.transform(X_train)
X_test_ref = erf.transform(X_val_and_test)
```

```
X_train_erf.shape
```

```
(736914, 4)
```

```
y.shape
```

```
(921143,)
```

Figure 10: Embedded Random Forest feature selection method.

Neural Network:

In this part we test the efficiency of examined feature selection methods in terms of overfitting and validation loss. To test it we require either an ML model or a Neural Network, my choice was on the second one. Firstly, we need to split the validation/test datasets into two separate ones, in this stage of research we only need the validation ones, so we will keep test sets for future:

```
# validation and test sets for chi method

X_test, X_val, Y_test, Y_val = train_test_split(X_test_chi, Y_val_and_test, test_size=0.2, random_state=1)

print(X_test.shape, X_val.shape, Y_test.shape, Y_val.shape)
```

```
(147383, 14) (36846, 14) (147383,) (36846,)
```

Figure 11: Example of data split on validation and test datasets.

Now, we need to create the model, as in this example we do not need the best performance, we create a 4 layer Sequential model, first two layers have 32 neurons and relu activation function, which is usually good in classification models. To slightly decrease overfitting, a dropout layer was added, as we have a big dataset dropping some values will decrease the chance of learning unneeded patterns, yet to keep the model overfitted no kernel regularization was added. As an output layer, we have a one value outcome, followed by a sigmoid function.

To compile the model we use 'sgd' as an optimizer, binnary crossentropy as a loss function. For now as the only metric we use accuracy, for future stages of research we will need more metrics to bild the best possible model

Finally, we train the model using the train datasets and validate it using the validation sets:

```
#creating model
model = Sequential([
    Dense(32, activation='relu', input_dim=14),
    Dense(32, activation='relu'),
    Dropout(0.25),
    Dense(1, activation='sigmoid'),
])
model.compile(optimizer='sgd',
              loss='binary_crossentropy',
              metrics=['accuracy'])
```

```
print(model.summary())
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 32)                480
_________________________________________________________________
dense_1 (Dense)              (None, 32)                1056
_________________________________________________________________
dropout (Dropout)            (None, 32)                0
_________________________________________________________________
dense_2 (Dense)              (None, 1)                 33
=================================================================
Total params: 1,569
Trainable params: 1,569
Non-trainable params: 0
_________________________________________________________________
None
```

```
#training model for chi method
h = model.fit(X_train_chi, Y_train,
          batch_size=10, epochs=50,
          validation_data=(X_val, Y_val))
```

Figure 12: Model creation and training.

The accuracy of the model seems incredibly high, in our case it is not a surprise as most of the outcomes are 0 and model can easily get high accuracy by just giving 0 to most of the samples, this challenge will be studied in the next stage of research.

For some cases we needed to change the input shape of first layers as some cases had features completely discarded like the Embedded Random Forest, that had only 4 features remaining.

The final step is to plot the validation loss for each case:

```
plt.plot(h.history['loss'])
plt.plot(h.history['val_loss'])
plt.title('Overall Loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'test'])
plt.show()
```

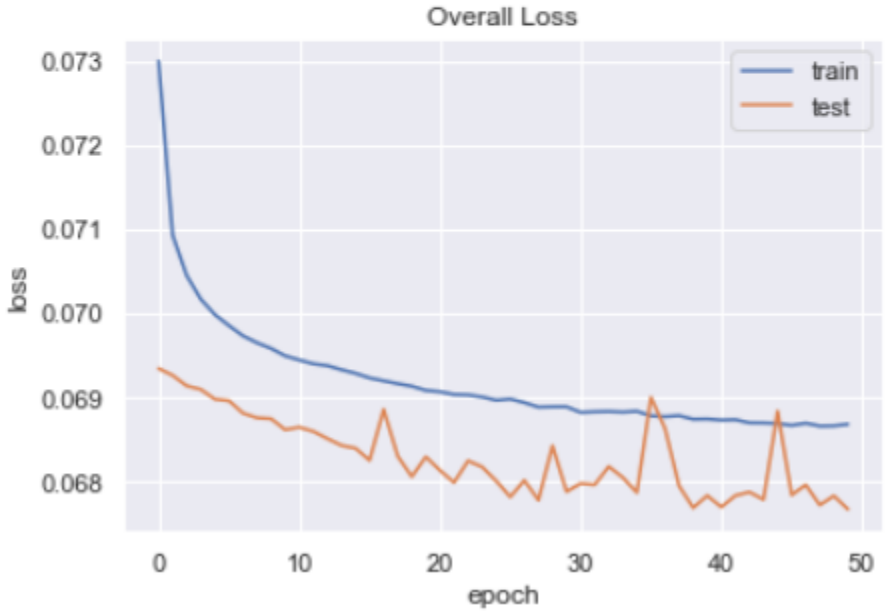


Figure 12: Validation loss for chi-square method.

```
plt.plot(h.history['loss'])
plt.plot(h.history['val_loss'])
plt.title('Overall Loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'test'])
plt.show()
```

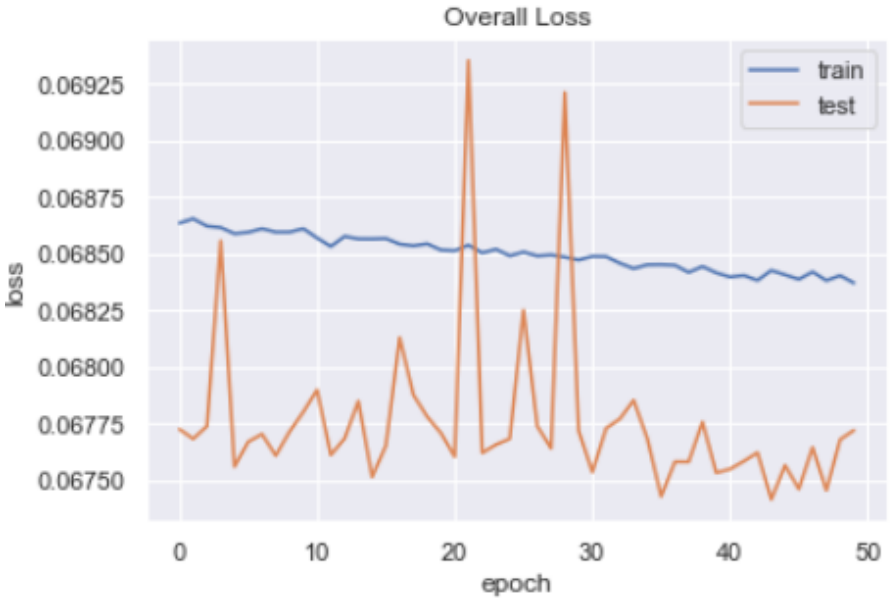


Figure 13: Validation loss for mutual information.

```
plt.plot(h2.history['loss'])
plt.plot(h2.history['val_loss'])
plt.title('Overall Loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'test'])
plt.show()
```

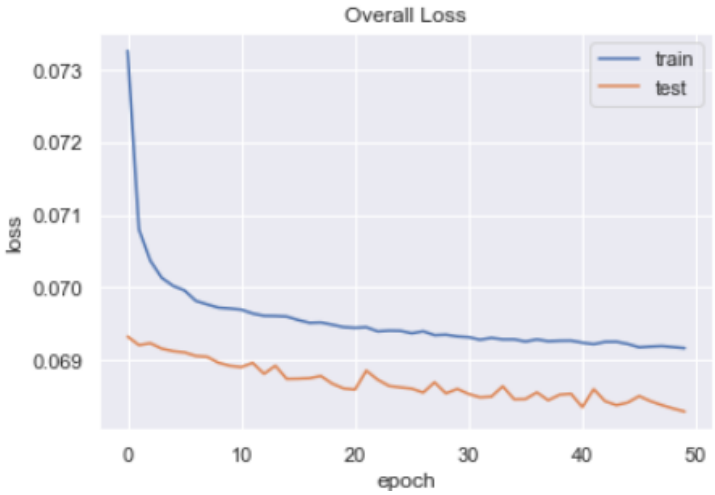
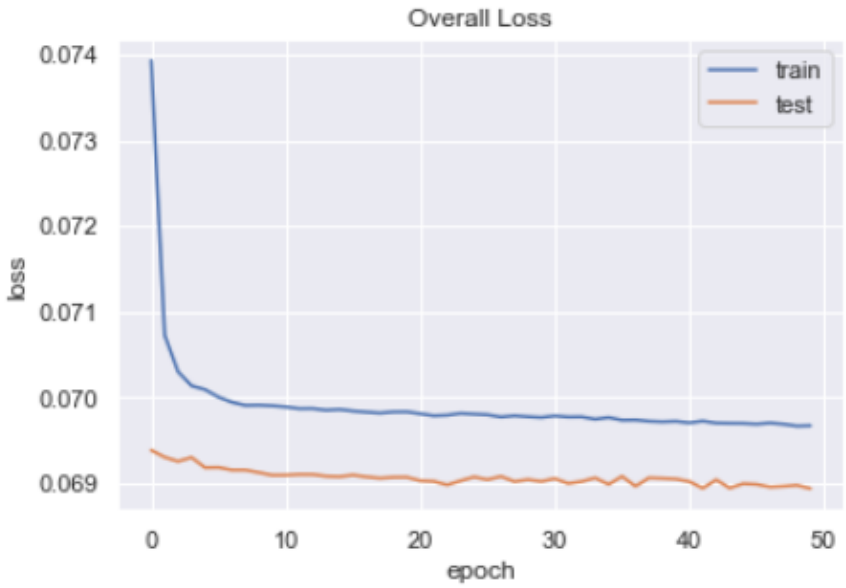


Figure 14: Validation loss for Recursive Feature Elimination.

```
plt.plot(h3.history['loss'])
plt.plot(h3.history['val_loss'])
plt.title('Overall Loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'test'])
plt.show()
```



Feature 15: Validation for Embedded Random Forest.

Result:

Analyzing the graphs, it can be seen that the Embedded Random Forest did the best job in fighting overfitting. It's validation graph looks better than those of the rest. Graph of Recursive Feature elimination does not look bad as well, but mutual information's is pretty terrible.

Conclusion:

Summarizing everything, my second hypothesis was correct, Embedded Random Forest deal well with mixed data, most likely by adding some kernel regularization, overfitting can be almost eliminated. In the next stages, I will upgrade the model and make it predict installations with high accuracy.