In [1]:

```
import pandas as pd

import matplotlib.pyplot as plt
%config InlineBackend.figure_format = 'retina'
```

In [83]:

```
data = pd.read_csv('Code_challenge_train.csv')
```

In [3]:

```
data.head(3)
```

Out[3]:

	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	...	x91	
0	-17.933519	6.559220	2.422468	-27.737392	-12.080601	-3.892934	1.067466	0.935953	10.912007	1.107144	...	11.107047	0.093
1	-37.214754	10.774930	5.404072	21.354738	0.612690	-3.093533	6.161558	-0.972156	-5.222169	0.384969	...	-1.991846	15.666
2	0.330441	-19.609972	-1.331804	-15.153892	19.710240	19.077300	-1.747110	0.545570	-1.464609	3.670570	...	17.132840	-5.333

3 rows × 101 columns

In [4]:

```
data.shape
```

Out[4]:

```
(40000, 101)
```

In [5]:

```
# How many of the variables doesn't have missing values?
nulls = data.isnull().sum().to_frame()
nulls.loc[nulls[0] == 0,:]
```

Out[5]:

	0
y	0

In [6]:

```
# how many missing at most?
nulls[0].sort_values(ascending = False).head()
```

Out[6]:

```
x55    16
x96    15
x21    15
x18    13
x63    13
Name: 0, dtype: int64
```

In [7]:

```
# percentage of missing values in the highest one
16 / data.shape[0] * 100
```

Out[7]:

```
0.04
```

In [8]:

```
# which variables are not numeric?
```

```
data.select_dtypes(include = 'object').head(3)
```

Out[8]:

	x34	x35	x41	x45	x68	x93
0	bmw	thur	$-1306.52	-0.01%	sept.	asia
1	Toyota	wednesday	$-24.86	0.0%	July	asia
2	bmw	thurday	$-110.85	0.0%	July	asia

Cleaning

Impute the nulls, in these object variables first

In [84]:

```
# 1. clean x41 and x45
data[['x41', 'x45']] = data[['x41', 'x45']].fillna('9999')
```

In [18]:

```
data[['x41', 'x45']].isnull().sum()
```

Out[18]:

```
x41    0
x45    0
dtype: int64
```

In [85]:

```
# strip the $ and %
data['x41'] = data['x41'].map(lambda x: x.strip('$'))
data['x45'] = data['x45'].map(lambda x: x.strip('%'))
```

In [86]:

```
data[['x41', 'x45']] = data[['x41', 'x45']].astype('float')
```

In [34]:

```
# 2. how many different values in each of the object variables?
object_columns = data.select_dtypes(include='object').columns.to_list()

for n in object_columns:
    print(f"{n}: {data[n].value_counts().shape[0]}")
```

```
x34: 10
x35: 8
x68: 12
x93: 3
```

In [33]:

```
# x35 should be 7 of less values.
data['x35'].value_counts(dropna = False)
```

Out[33]:

```
wed          14829
thurday      13323
wednesday     5927
thur          4403
tuesday        898
friday         528
monday          59
fri             22
NaN             11
Name: x35, dtype: int64
```

In [87]:

```
# before we start cleaning that one up, we need to impute missing values. I'll fill them with 'unknown'
data[object_columns] = data[object_columns].fillna('unknown')
```

In [39]:

```
data['x35'].value_counts(dropna = False)
```

Out[39]:

```
wed          14829
thurday      13323
wednesday     5927
thur          4403
tuesday        898
friday         528
monday          59
fri             22
unknown         11
Name: x35, dtype: int64
```

In [52]:

```
def weekdays(v):
    """
    This is a supporting function to cleaning(), to be used inside the latter.
    Meant to be applied to each value of the weekdays variable.
    """
    if 'wed' in v:
        v = 'wednesday'
    elif 'thu' in v:
        v = 'thursday'
    elif 'fri' in v:
        v = 'friday'

    return v
```

In [88]:

```
data['x35'] = data['x35'].map(lambda x: weekdays(x))
```

In [101]:

```
# Combining all cleaning steps in a function
def cleaning(data):
    """
    This function is specific to 'Code_challenge_train.csv' and 'Code_challenge_test.csv'

    Parameters:
    -----------
    data: the data frame to clean.
    """
    # numerical variables with strange characters and nulls: replace missing value with '9999'
    data[['x41', 'x45']] = data[['x41', 'x45']].fillna('9999')

    # strip the strange characters, and replace
    data['x41'] = data['x41'].map(lambda x: x.strip('$'))
    data['x45'] = data['x45'].map(lambda x: x.strip('%'))
    data[['x41', 'x45']] = data[['x41', 'x45']].astype('float')

    # filling nulls in rest of string variables with 'unknown'
    object_columns = data.select_dtypes(include='object').columns.to_list()
    data[object_columns] = data[object_columns].fillna('unknown')

    # weekdays variable is messy, cleaning it
    data['x35'] = data['x35'].map(lambda x: weekdays(x))

    # dropping 'x45'
    data.drop(['x45'], axis = 1, inplace = True)

    # changing months to numbers
    months_dict = {'January': 1, 'Feb': 2, 'Mar':3, 'Apr': 4, 'May': 5, 'Jun': 6, 'July': 7, 'Aug': 8,
'sept.': 9,
                    'Oct': 10, 'Nov': 11, 'Dev': 12, 'unknown': 0}
    data['x68'] = data['x68'].map(months_dict)

    # one-hot encode the other string variables
    string_variables = data.select_dtypes(include='object').columns.to_list()
    string_dummies = pd.get_dummies(data[string_variables])
    data = pd.concat([data, string_dummies], axis = 1, sort = False)

    # dropping extra columns
```

```python
    # dropping extra columns
    unknowns_columns = [n for n in data.columns if 'unknown' in n]
    data.drop(unknowns_columns, axis = 1, inplace = True)
    data.drop(['x34', 'x35', 'x93'], axis = 1, inplace = True)

    # filling missing values in the numerical variables
    numerical_columns = data.select_dtypes(exclude = 'object').columns.to_list()
    data[numerical_columns] = data[numerical_columns].fillna(method = 'bfill')

    return data
```

In [53]:

```python
# we need to fill all missing values,
# but first, let's see the distribution of these numerical variables, and if there's outliers
data.hist(figsize = (25, 25));
```

Observations:
All the numerical variables are roughly normal. Except for 'x45', which is can be safely dropped. because it's almost the same always. Note: we imputed the nulls with 9999 so it's not really outliers that it has
'x41' has a right outlier. 'x26', and 'x16' have moderate left outliers

In [57]:

```python
data['x45'].value_counts()
```

Out[57]:

```
0.00        15524
```

```
 0.01        9577
-0.01        9569
 0.02        2390
-0.02        2374
-0.03         279
 0.03         257
-0.04          14
 0.04          11
 9999.00        5
Name: x45, dtype: int64
```

Before dropping the variable, let's double check that it doesn't have strong correlation with the target variable, as this would be explanatory.

In [64]:

```
data['x45'].corr(data['y'])
```

Out[64]:

```
-9.146712948342792e-05
```

In [65]:

```
# ok, maybe there's no linear correlation, but how about its second degree?
data['x45'].map(lambda x: x ** 2).corr(data['y'])
```

Out[65]:

```
-8.3392344943707e-05
```

We're good to drop it

In [89]:

```
data.drop(['x45'], axis = 1, inplace = True)
```

Imputing missing values from the rest of the numerical variables:
since there isn't many missing values, imputing them won't affect results badly.
We have at most 16 missing values, and they are spread out well enough to fill the missing values with 'bfill' or 'pad' rather than 'interpolate' which we can also do.
The code used to check these values is:

```
import numpy as np
for n in data.select_dtypes(exclude = 'object').columns.to_list():
    print(n)
    print(np.nonzero(pd.isnull(data[n])))
    print('---------')
```

I didn't want to clutter the notebook, so I ran it and deleted the cell.

In [100]:

```
numerical_columns = data.select_dtypes(exclude = 'object').columns.to_list()
data[numerical_columns] = data[numerical_columns].fillna(method = 'bfill')
```

Now to change the categorical variables to numericals

In [132]:

```
# recall that x68 was the months. Let's take care of that first
months_dict = {'January': 1, 'Feb': 2, 'Mar':3, 'Apr': 4, 'May': 5, 'Jun': 6, 'July': 7, 'Aug': 8, 'sept.': 9,
'Oct': 10, 'Nov': 11, 'Dev': 12, 'unknown': 0}

data['x68'] = data['x68'].map(months_dict)
```

In [135]:

```
string_variables = data.select_dtypes(include='object').columns.to_list()
string_dummies = pd.get_dummies(data[string_variables])
data = pd.concat([data, string_dummies], axis = 1, sort = False)
```

In [140]:

```
unknowns_columns = [n for n in data.columns if 'unknown' in n]
data.drop(unknowns_columns, axis = 1, inplace = True)
# no information is lost here.
```

Alternatively, we could've one-hot encode the string variables first, including the nulls, then drop the null columns

In [141]:

```
data.shape
```

Out[141]:

```
(40000, 118)
```

In [102]:

```
# double checking one final time that we don't have any nulls
nulls = data.isnull().sum().to_frame()
nulls.loc[nulls[0] != 0, :]
```

Out[102]:

0

```
import pandas as pd

def weekdays(v):
    """
    This is a supporting function to cleaning(), to be used inside the
latter.
    Meant to be applied to each value of the weekdays variable.
    """
    if 'wed' in v:
        v = 'wednesday'
    elif 'thu' in v:
        v = 'thursday'
    elif 'fri' in v:
        v = 'friday'

    return v


# Combining all cleaning steps in a function
def cleaning(data):
    """
    This function is specific to 'Code_challenge_train.csv' and
'Code_challenge_test.csv'

    Parameters:
    -----------
    data: the data frame to clean.
    """
    # numerical variables with strange characters and nulls: replace
missing value with '9999'
    data[['x41', 'x45']] = data[['x41', 'x45']].fillna('9999')

    # strip the strange characters, and replace
    data['x41'] = data['x41'].map(lambda x: x.strip('$'))
    data['x45'] = data['x45'].map(lambda x: x.strip('%'))
    data[['x41', 'x45']] = data[['x41', 'x45']].astype('float')

    # filling nulls in rest of string variables with 'unknown'
    object_columns =
data.select_dtypes(include='object').columns.to_list()
    data[object_columns] = data[object_columns].fillna('unknown')

    # weekdays variable is messy, cleaning it
    data['x35'] = data['x35'].map(lambda x: weekdays(x))

    # dropping 'x45'
    data.drop(['x45'], axis = 1, inplace = True)

    # changing months to numbers
    months_dict = {'January': 1, 'Feb': 2, 'Mar':3, 'Apr': 4, 'May':
```

```
5, 'Jun': 6, 'July': 7, 'Aug': 8, 'sept.': 9,
                    'Oct': 10, 'Nov': 11, 'Dev': 12, 'unknown': 0}
    data['x68'] = data['x68'].map(months_dict)

    # one-hot encode the other string variables
    string_variables =
data.select_dtypes(include='object').columns.to_list()
    string_dummies = pd.get_dummies(data[string_variables])
    data = pd.concat([data, string_dummies], axis = 1, sort = False)

    # dropping extra columns
    unknowns_columns = [n for n in data.columns if 'unknown' in n]
    data.drop(unknowns_columns, axis = 1, inplace = True)
    data.drop(['x34', 'x35', 'x93'], axis = 1, inplace = True)

    # filling missing values in the numerical variables
    numerical_columns = data.select_dtypes(exclude =
'object').columns.to_list()
    data[numerical_columns] = data[numerical_columns].fillna(method =
'bfill')

    return data
```

In [181]:
```
import pandas as pd

import matplotlib.pyplot as plt
%config InlineBackend.figure_format = 'retina'
```

In [182]:
```
import cleaning_functions as clean
```

In [183]:
```
data = pd.read_csv('Code_challenge_train.csv')
```

In [184]:
```
data.shape
```

Out[184]: (40000, 101)

In [185]:
```
data.head(3)
```

Out[185]:

	x0	x1	x2	x3	x4	x5	x6	x7	
0	-17.933519	6.559220	2.422468	-27.737392	-12.080601	-3.892934	1.067466	0.935953	10.912(
1	-37.214754	10.774930	5.404072	21.354738	0.612690	-3.093533	6.161558	-0.972156	-5.222
2	0.330441	-19.609972	-1.331804	-15.153892	19.710240	19.077300	-1.747110	0.545570	-1.464(

3 rows × 101 columns

In [186]:
```
data = clean.cleaning(data)
```

In [187]:
```
data.shape
```

Out[187]: (40000, 115)

In [188]:
```
nulls = data.isnull().sum().to_frame()
nulls.loc[nulls[0] != 0, :]
```

Out[188]:

0

In [189]:
```
data.head(3)
```

Out[189]:

	x0	x1	x2	x3	x4	x5	x6	x7	
0	-17.933519	6.559220	2.422468	-27.737392	-12.080601	-3.892934	1.067466	0.935953	10.912(
1	-37.214754	10.774930	5.404072	21.354738	0.612690	-3.093533	6.161558	-0.972156	-5.222
2	0.330441	-19.609972	-1.331804	-15.153892	19.710240	19.077300	-1.747110	0.545570	-1.464(

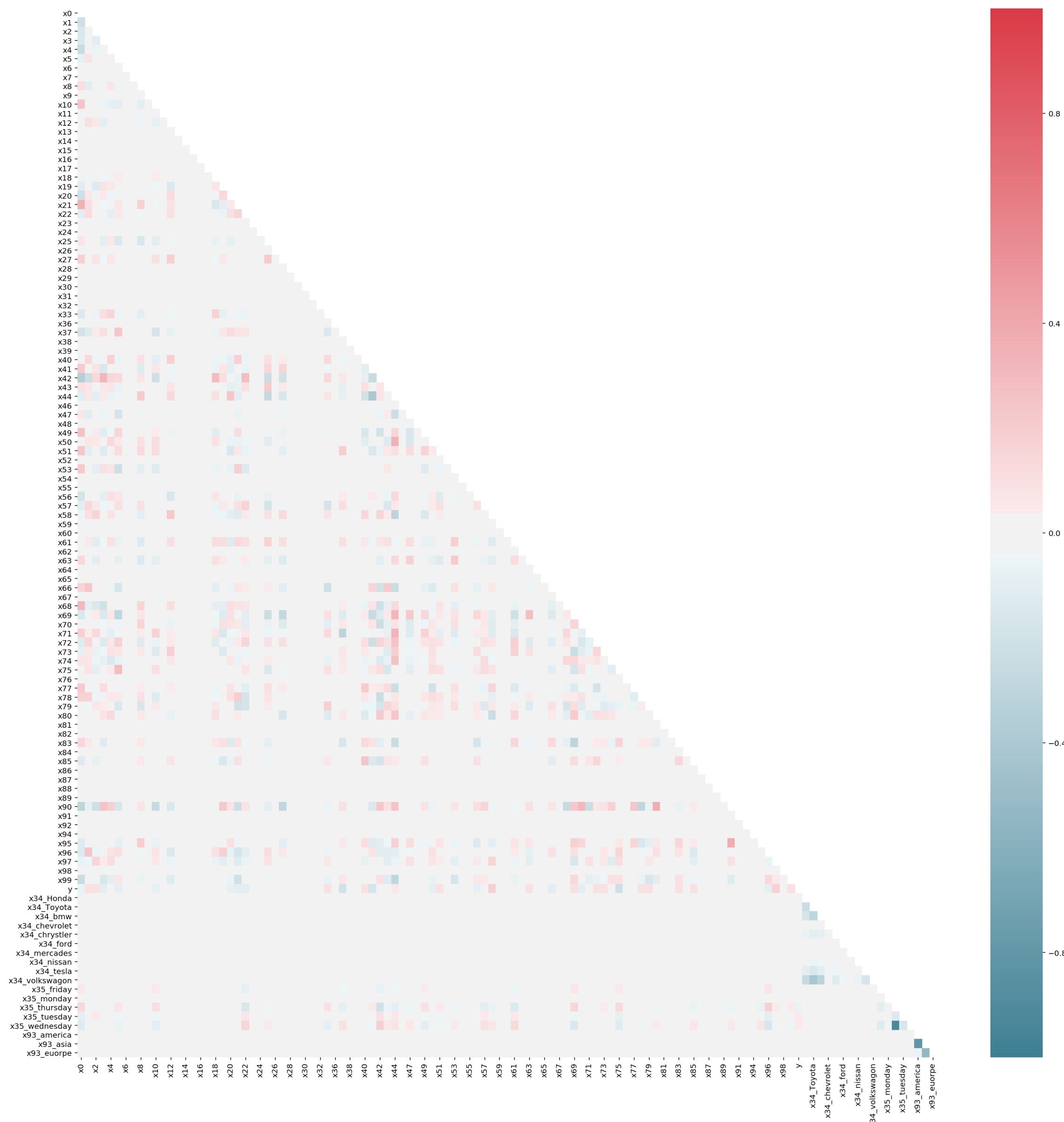
3 rows × 115 columns

Exploring Correlations

In [190]:
```
def make_heat_map(df = data):
```

```python
    """
    Plots the heatmap of correlations of the given df
    """
    import seaborn as sns
    import numpy as np

    mask = np.zeros_like(df.corr())
    mask[np.triu_indices_from(mask)] = True

    plt.figure(figsize = (25,25))

    cmap = sns.diverging_palette(220, 10, as_cmap=True)

    sns.heatmap(df.corr(), mask = mask, vmin = -1, vmax = 1, cmap = cmap);
```

In [191]:
```python
make_heat_map(data);
```



Observations:
No strong linear correlations with the target variable, or within the variables. But that doesn't mean there is no correlation at all, there might be non-linear relationships. I will see later if I have squared relationships between the target variable, and the second degree variables, and their interactions.

At least, it is good to know there's no multi-collinearity in the dataset, meaning I don't need to use Principal Component Analysis, as it is not a good fit for this dataset.

The colinearity between the one-hot encoded variables of the same original one is expected, and doesn't affect what I observed.

Creating a testing set to measure performance of models, and continue with the explorations

In [192]:
```
data['y'].value_counts()
```

Out[192]:
```
0    31880
1     8120
Name: y, dtype: int64
```

In [193]:
```
data['y'].value_counts(normalize = True)
```

Out[193]:
```
0    0.797
1    0.203
Name: y, dtype: float64
```

In [194]:
```
from sklearn.model_selection import train_test_split

X = data.drop('y', axis = 1)
y = data['y']

X_train, X_test, y_train, y_test = train_test_split(X, y, stratify = y, rand
om_state = 72019)
```

In [195]:
```
from sklearn.preprocessing import StandardScaler, PolynomialFeatures

ss = StandardScaler()

X_train = ss.fit_transform(X_train)
X_test = ss.transform(X_test)
```

```
/anaconda3/lib/python3.6/site-packages/sklearn/preprocessing/data.py:625:
DataConversionWarning: Data with input dtype uint8, int64, float64 were a
ll converted to float64 by StandardScaler.
  return self.partial_fit(X, y)
/anaconda3/lib/python3.6/site-packages/sklearn/base.py:462: DataConversio
nWarning: Data with input dtype uint8, int64, float64 were all converted
to float64 by StandardScaler.
  return self.fit(X, **fit_params).transform(X)
/anaconda3/lib/python3.6/site-packages/ipykernel_launcher.py:6: DataConve
rsionWarning: Data with input dtype uint8, int64, float64 were all conver
ted to float64 by StandardScaler.
```

In [196]:
```
poly = PolynomialFeatures(degree = 2, interaction_only = False)
```

In [197]:
```
X_train_poly = poly.fit_transform(X_train)
X_test_poly = poly.transform(X_test)
```

In [198]:
```
X_train_poly.shape
```

Out[198]:
```
(30000, 6670)
```

In [199]:
```
corr_poly = pd.DataFrame(X_train_poly).corrwith(y)
```

In [200]:
```python
correlations = corr_poly.to_frame(name = 'poly_corrs')
```

In [201]:
```python
correlations['abs_value'] = correlations['poly_corrs'].abs()
```

In [202]:
```python
correlations.sort_values(by = 'abs_value', ascending = False).head()
```

Out[202]:

	poly_corrs	abs_value
2820	0.021882	0.021882
1137	-0.021647	0.021647
3373	-0.021641	0.021641
1697	-0.020895	0.020895
6176	-0.020552	0.020552

That wouldn't add any explanatory value to the model, because correlations with y still almost don't exist

In [203]:
```python
data.select_dtypes(include='object').shape
# we have all numeric variables now, and we are set to train models in the n
ext notebook
```

Out[203]:
```
(40000, 0)
```

So far, and next steps:
We have very imbalanced classes, instead of upping the positive class, or down-sampling the negative class. I'm gonna try SVM, Random Forests and XGBoost, although I know already that XGBoost will perform best.
Measures: AUC. I will add f1-score and confusion matrix for a full picture.

Some visualizations

going back to before one-hot encoding, because I want to group mean of y based on the string variables. I want to see if there's some sort of pattern. If for specific makes in a certain content, for example, are more label 1, etc.

In [207]:
```python
def weekdays(v):
    """
    This is a supporting function to cleaning(), to be used inside the latte
r.
    Meant to be applied to each value of the weekdays variable.
    """
    if 'wed' in v:
        v = 'wednesday'
    elif 'thu' in v:
        v = 'thursday'
    elif 'fri' in v:
        v = 'friday'

    return v
```

```python
# Combining all cleaning steps in a function
def cleaning(data):
    """
    This function is specific to 'Code_challenge_train.csv' and 'Code_challe
nge_test.csv'

    Parameters:
    -----------
    data: the data frame to clean.
    """
    # numerical variables with strange characters and nulls: replace missing
value with '9999'
    data[['x41', 'x45']] = data[['x41', 'x45']].fillna('9999')

    # strip the strange characters, and replace
    data['x41'] = data['x41'].map(lambda x: x.strip('$'))
    data['x45'] = data['x45'].map(lambda x: x.strip('%'))
    data[['x41', 'x45']] = data[['x41', 'x45']].astype('float')

    # filling nulls in rest of string variables with 'unknown'
    object_columns = data.select_dtypes(include='object').columns.to_list()
    data[object_columns] = data[object_columns].fillna('unknown')

    # weekdays variable is messy, cleaning it
    data['x35'] = data['x35'].map(lambda x: weekdays(x))

    # dropping 'x45'
    data.drop(['x45'], axis = 1, inplace = True)

    # changing months to numbers
    months_dict = {'January': 1, 'Feb': 2, 'Mar':3, 'Apr': 4, 'May': 5, 'Ju
n': 6, 'July': 7, 'Aug': 8, 'sept.': 9,
                   'Oct': 10, 'Nov': 11, 'Dev': 12, 'unknown': 0}
    data['x68'] = data['x68'].map(months_dict)

#     # one-hot encode the other string variables
#     string_variables = data.select_dtypes(include='object').columns.to_lis
t()
#     string_dummies = pd.get_dummies(data[string_variables])
#     data = pd.concat([data, string_dummies], axis = 1, sort = False)

#     # dropping extra columns
#     unknowns_columns = [n for n in data.columns if 'unknown' in n]
#     data.drop(unknowns_columns, axis = 1, inplace = True)
#     data.drop(['x34', 'x35', 'x93'], axis = 1, inplace = True)

    # filling missing values in the numerical variables
    numerical_columns = data.select_dtypes(exclude = 'object').columns.to_li
st()
    data[numerical_columns] = data[numerical_columns].fillna(method = 'bfil
l')

    return data
```

In [208]:
```python
data = pd.read_csv('Code_challenge_train.csv')
```

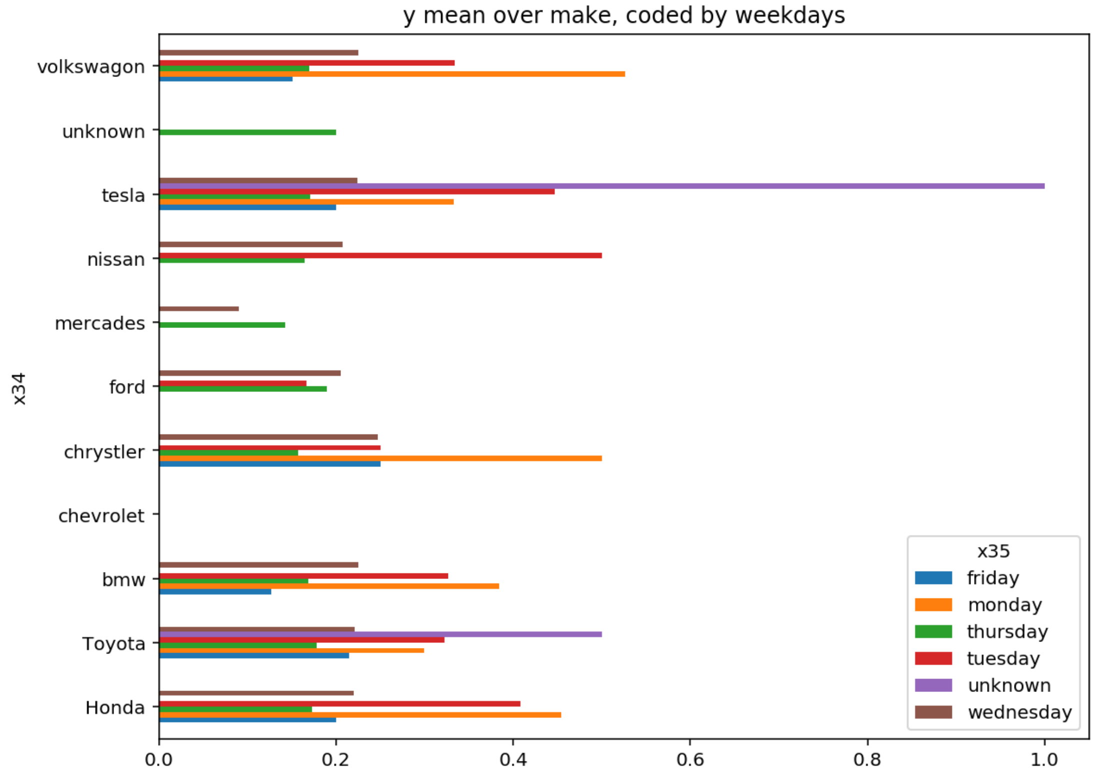
In [209]:
```python
data = cleaning(data)
```

In [210]:
```python
data.select_dtypes(include='object').head(1)
```

Out[210]:

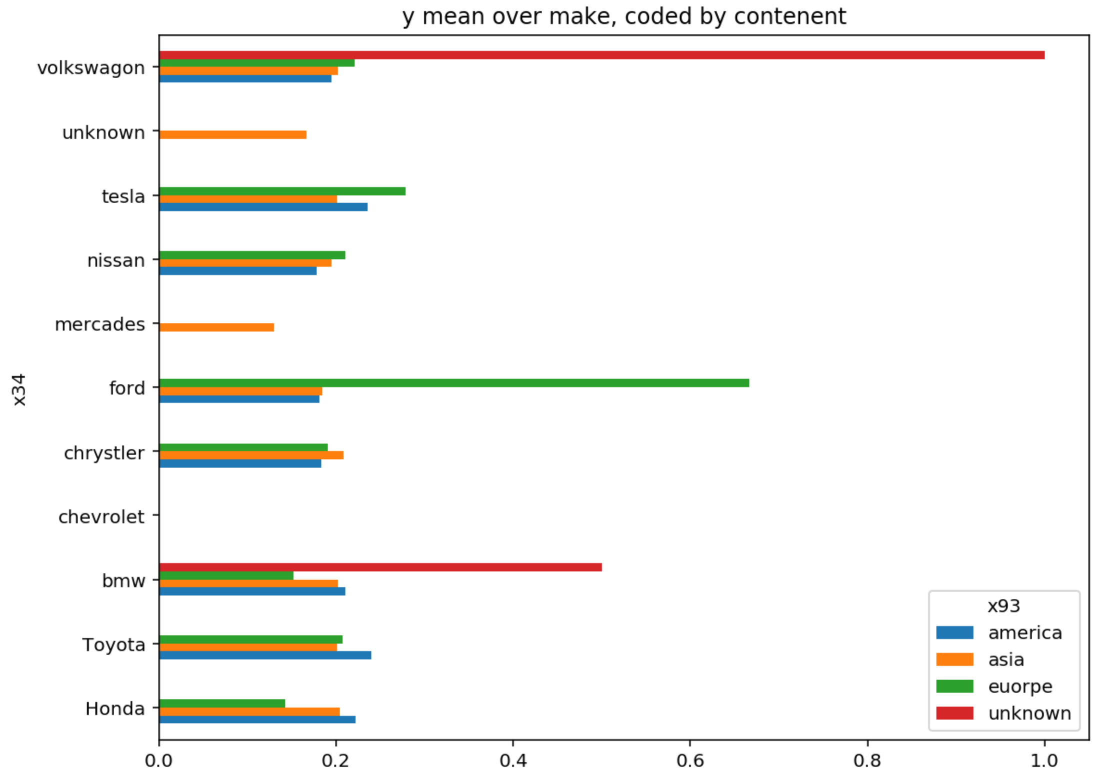
	x34	x35	x93
0	bmw	thursday	asia

In [211]:
```
data.groupby(['x34', 'x35'])['y'].mean().unstack().plot(kind = 'barh', figsize = (9,7),
                                                         title = 'y mean over make, coded by weekdays');
```



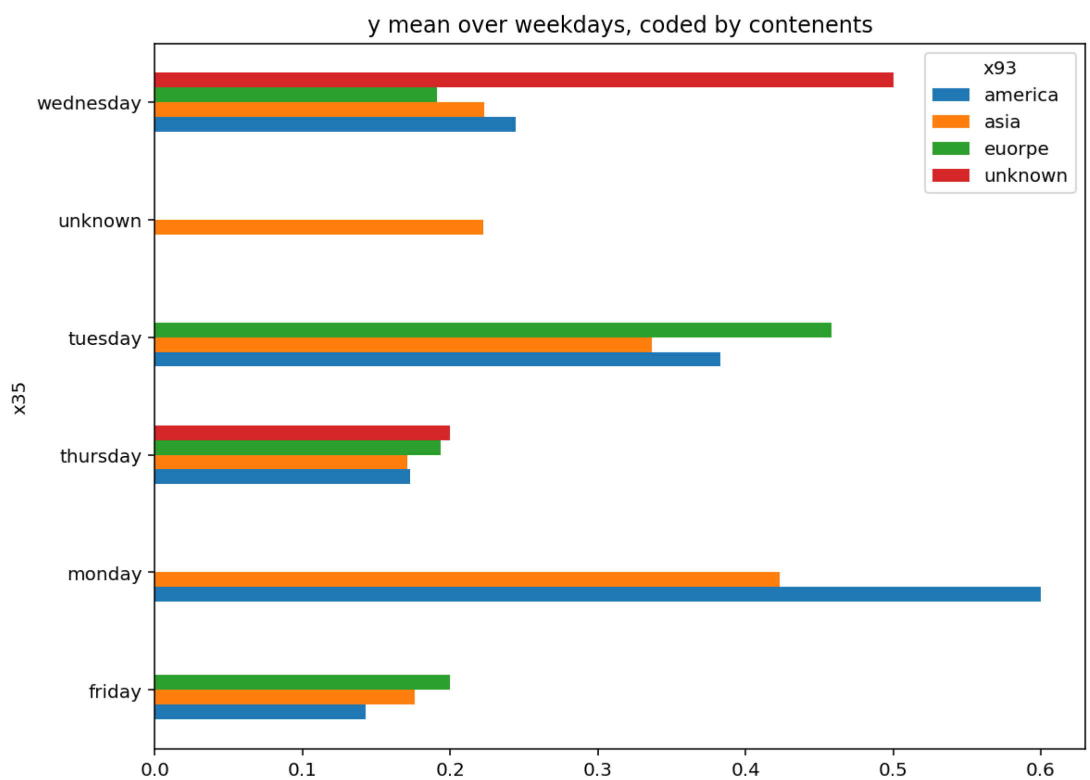
Not all visualizations are going to be insightful. for example, this one tells me that a lot of Tesla on some unknown day, are labeled 1.
of course knowing the dictionary of the data, and what each variable means, I can draw better conclusions

In [212]:
```
data.groupby(['x34', 'x93'])['y'].mean().unstack().plot(kind = 'barh', figsize = (9,7),
                                                         title = 'y mean over make, coded by contenent');
```



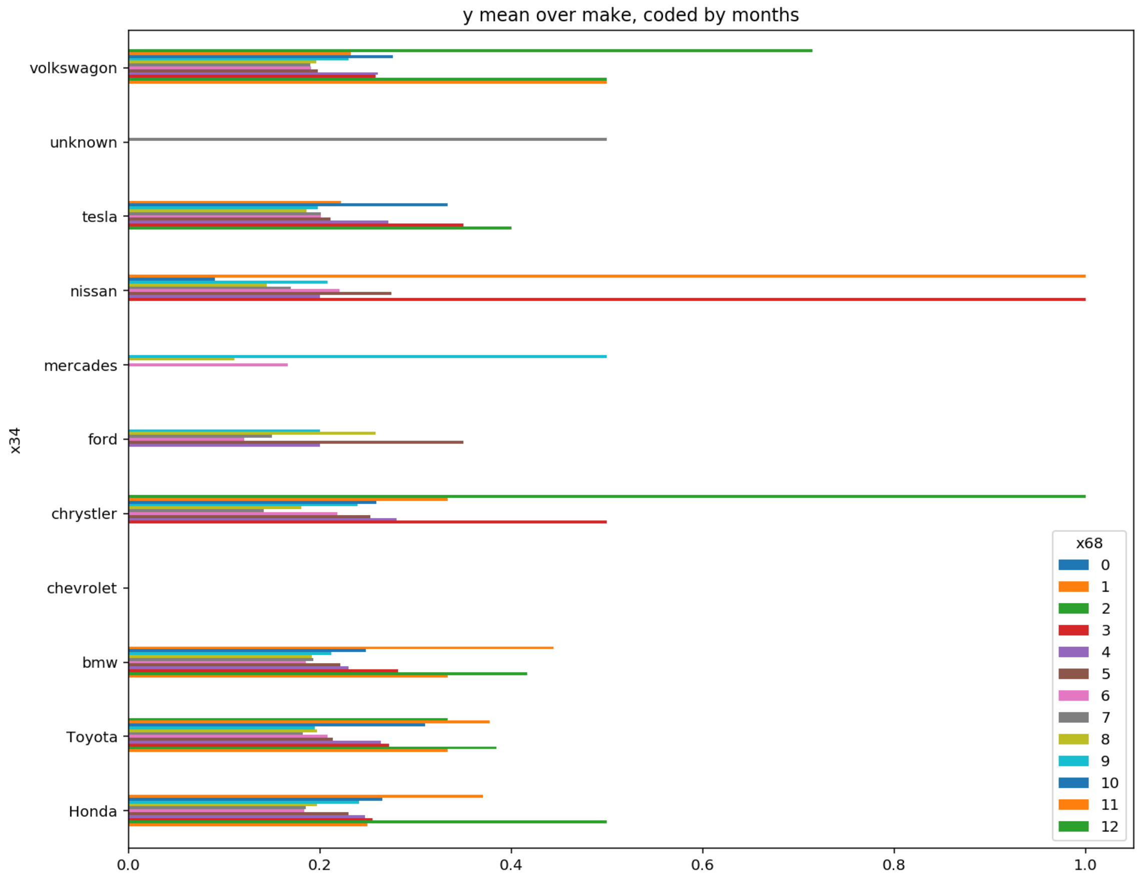
Same issue here. Volkswagon in some unknown contenent was labeled 1 a lot. Followed by Ford in Europe, and BMW in some unknown contenent.

In [213]:
```
data.groupby(['x35', 'x93'])['y'].mean().unstack().plot(kind = 'barh', figsize = (9,7),
                                                         title = 'y mean over weekdays, coded by contenents');
```



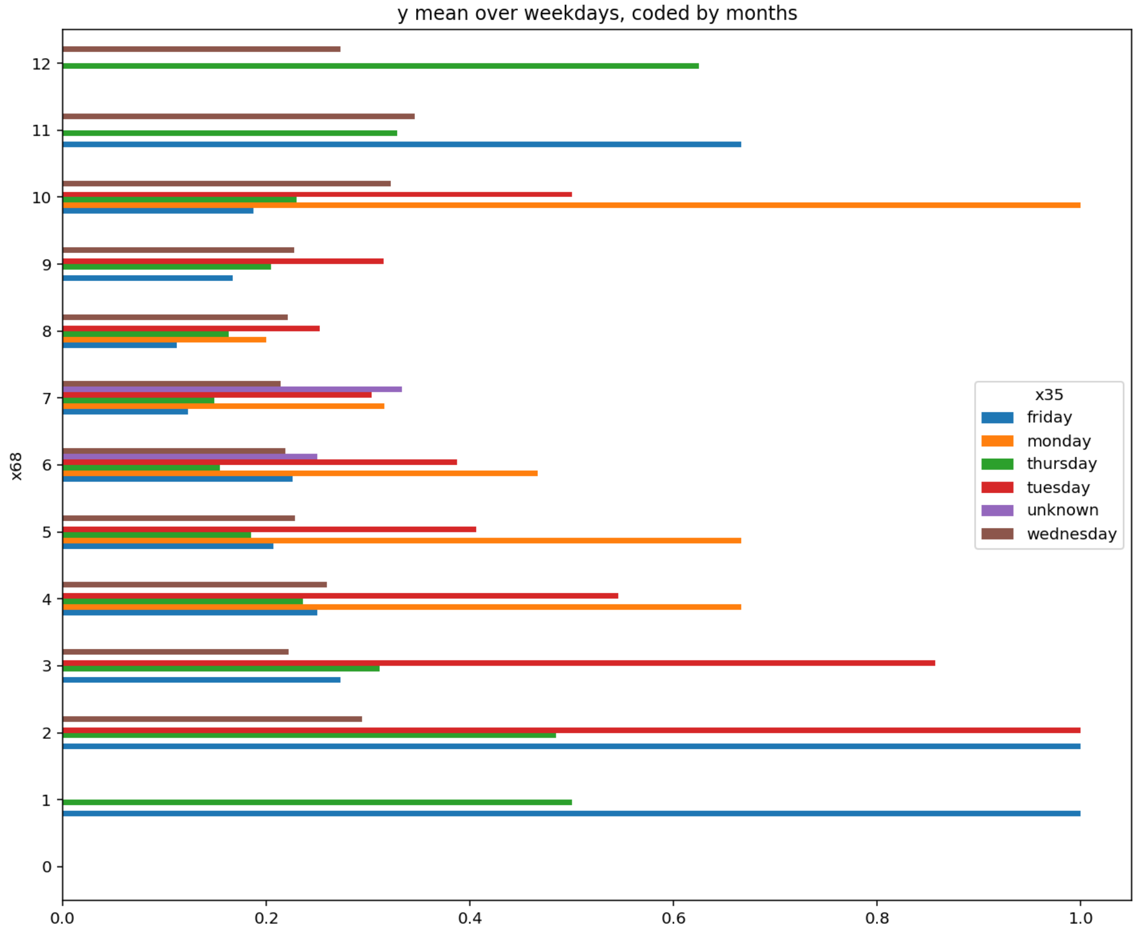
America on Mondays and Tuesdays was labeled 1 more often than 0, followed by Europe on Tuesdays

In [214]:
```
data.groupby(['x34', 'x68'])['y'].mean().unstack().plot(kind = 'barh', figsize = (12,10),
                                                        title = 'y mean over make, coded by months');
```



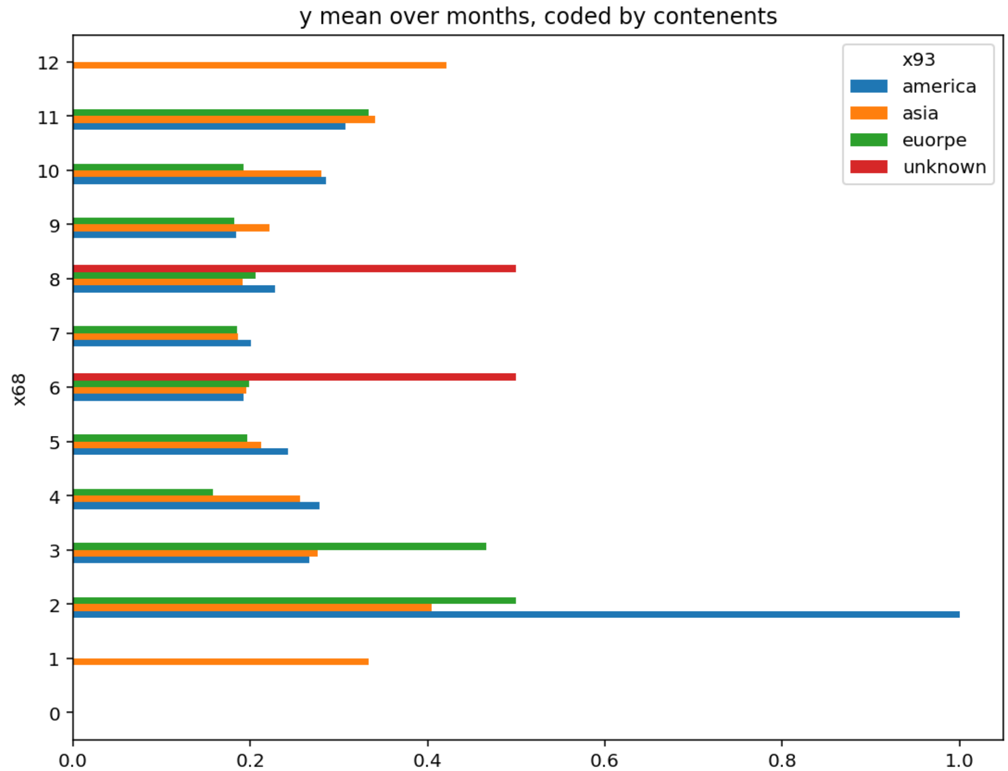
This is a hard to read plot, making it not particularly a good one. Although we can tell clearly on first scan, is that Nissan was labeled 1 a whole lot in January, and March, followed by Chrystler in February, then Volkswagon in Feb. as well

In [215]:
```
data.groupby(['x68', 'x35'])['y'].mean().unstack().plot(kind = 'barh', figsize = (12,10),
                                                        title = 'y mean over weekdays, coded by months');
```



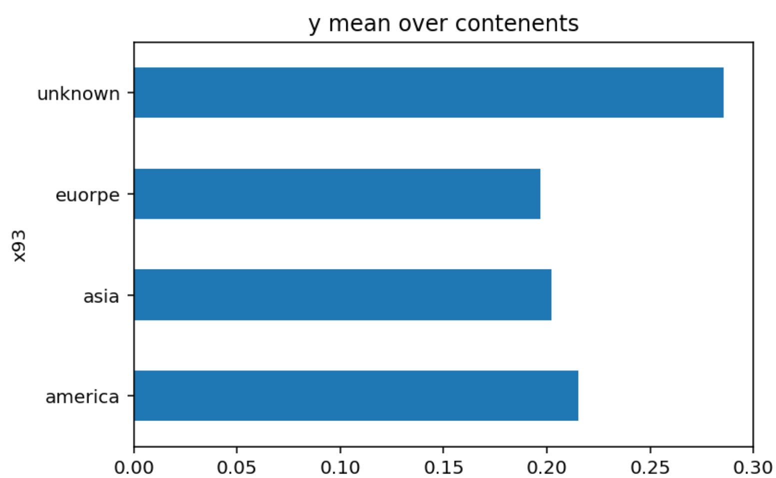
More positive labels occured on Fridays during January and February, on Mondays during October. On Tuesdays during Feb and March. The rest are either equally spread between the two labels, or more frequently belonged to the negative class.

In [216]:
```
data.groupby(['x68', 'x93'])['y'].mean().unstack().plot(kind = 'barh', figsi
ze = (9,7),
                                                        title = 'y mean over
 months, coded by contenents');
```



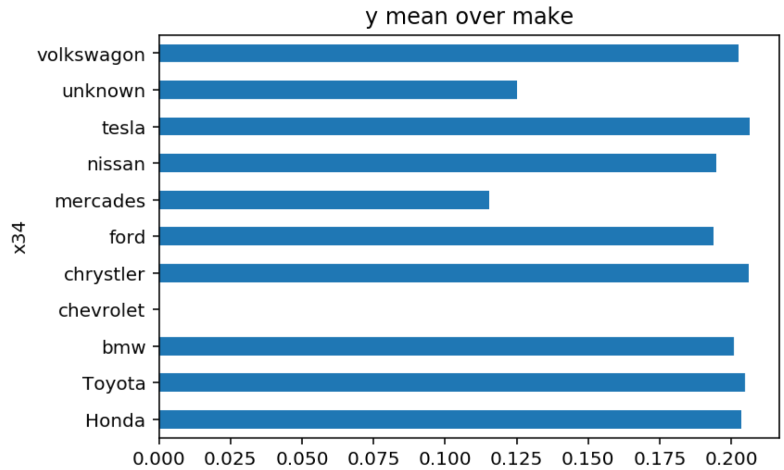
One stands out, America in February has more positive class by a big difference than all the rest.

In [217]:
```
data.groupby(['x93'])['y'].mean().plot(kind = 'barh', title = 'y mean over c
ontenents');
```



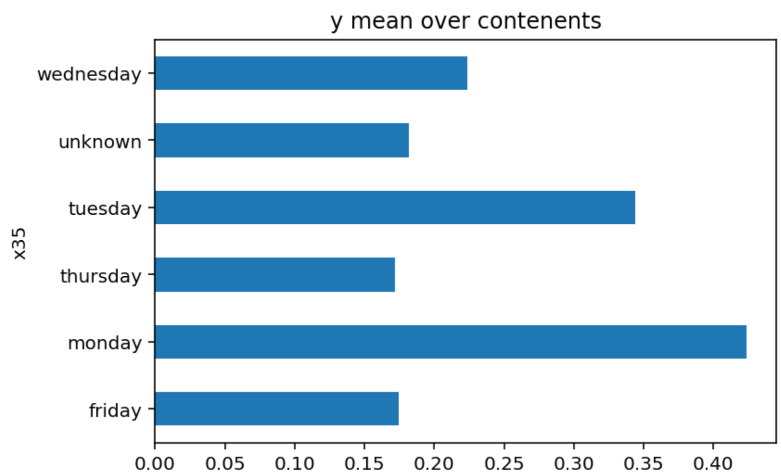
Individually speaking, the contents shared almost the same frequency between the two classes, holding everything else fixed.

In [218]:
```
data.groupby(['x34'])['y'].mean().plot(kind = 'barh', title = 'y mean over m
ake');
```



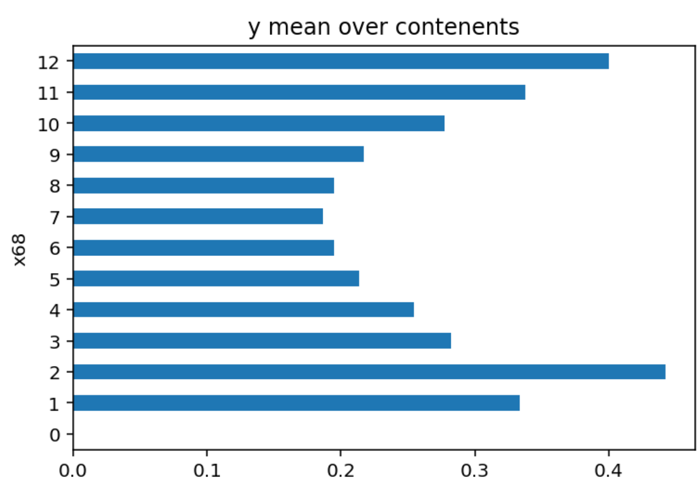
Same conclusion for car make

In [219]:
```
data.groupby(['x35'])['y'].mean().plot(kind = 'barh', title = 'y mean over c
ontenents');
```



We can see some distinctions here, beginning of the week has more positive class labels than the rest of the week.

In [220]:
```
data.groupby(['x68'])['y'].mean().plot(kind = 'barh', title = 'y mean over c
ontenents');
```



There's seasonality in this data. More cases (observations) carry the negative label during summer, and picks up during winter months, specifically February and December, were observations caried the positive label more frequenlty.

In [ ]:

In [1]:

```
import pandas as pd

import matplotlib.pyplot as plt
%config InlineBackend.figure_format = 'retina'

import cleaning_functions as clean
```

In [2]:

```
data = pd.read_csv('Code_challenge_train.csv')
```

In [3]:

```
data = clean.cleaning(data)
```

In [4]:

```
nulls = data.isnull().sum().to_frame()
nulls.loc[nulls[0] != 0, :]
```

Out[4]:

0

In [5]:

```
from sklearn.model_selection import train_test_split

X = data.drop('y', axis = 1)
y = data['y']

X_train, X_test, y_train, y_test = train_test_split(X, y, stratify = y, random_state = 72019)
```

In [7]:

```
from sklearn.preprocessing import StandardScaler, MinMaxScaler
from sklearn.feature_selection import SelectKBest
from sklearn.decomposition import PCA
from sklearn.ensemble import RandomForestClassifier
from sklearn.svm import SVC
from sklearn.naive_bayes import MultinomialNB, GaussianNB
from sklearn.neighbors import KNeighborsClassifier
from sklearn.linear_model import SGDClassifier

from xgboost import XGBClassifier, XGBRFClassifier

from sklearn.pipeline import Pipeline
from sklearn.model_selection import GridSearchCV, RandomizedSearchCV
```

I will run through a few models quickly, using their default hyperparameters, and the full dataset, to get a rough estimate of the performance of each. Then I will invest my time in fine-tuning the model, and further feature engineering

In [8]:

```
def run_model(name):

    X_train, X_test, y_train, y_test = train_test_split(X, y, stratify = y, random_state = 72019)

    if name not in [MultinomialNB, GaussianNB]:

        ss = StandardScaler()
        X_train = ss.fit_transform(X_train)
        X_test = ss.transform(X_test)

        model = name()

        model.fit(X_train, y_train)
```

```
        scores = model.score(X_test, y_test)

        return scores

    else:
        mm = MinMaxScaler()
        X_train = mm.fit_transform(X_train)
        X_test = mm.transform(X_test)

        model = name()
        model.fit(X_train, y_train)

        scores = model.score(X_test, y_test)

        return scores
```

In [9]:

```
import warnings
warnings.filterwarnings('ignore')
```

In [10]:

```
model_list = [MultinomialNB, GaussianNB, RandomForestClassifier, SVC, KNeighborsClassifier, SGDClassifi
er,
              XGBClassifier, XGBRFClassifier]
for m in model_list:
    print(m)
    print(run_model(m))
    print('--------')
```

```
<class 'sklearn.naive_bayes.MultinomialNB'>
0.797
--------
<class 'sklearn.naive_bayes.GaussianNB'>
0.2474
--------
<class 'sklearn.ensemble.forest.RandomForestClassifier'>
0.859
--------
<class 'sklearn.svm.classes.SVC'>
0.9832
--------
<class 'sklearn.neighbors.classification.KNeighborsClassifier'>
0.9111
--------
<class 'sklearn.linear_model.stochastic_gradient.SGDClassifier'>
0.8471
--------
<class 'xgboost.sklearn.XGBClassifier'>
0.9041
--------
<class 'xgboost.sklearn.XGBRFClassifier'>
0.8123
--------
```

Looks like we have a winner. We have very imbalanced classes, I thought it would be either SVM or boosted random forests. I will still run a quick XGBoost just because I want to make sure, as SVM is slow, tuning it would take more resources

In [10]:

```
anova_svc = Pipeline([
    ('ss' , StandardScaler()),
    ('anova' , SelectKBest()),
    ('svc', SVC())
])
```

In [11]:

```
anova_svc.fit(X_train, y_train)
```

Out[11]:

```
Pipeline(memory=None,
     steps=[('ss', StandardScaler(copy=True, with_mean=True, with_std=True)), ('anova', SelectKBest(k=1
0, score_func=<function f_classif at 0x1239c6730>)), ('svc', SVC(C=1.0, cache_size=200, class_weight=No
```

```
0, score_func=<function f_classif at 0x1259e0730>)), ('svc', SVC(C=1.0, cache_size=200, class_weight=No
ne, coef0=0.0,
  decision_function_shape='ovr', degree=3, gamma='auto_deprecated',
  kernel='rbf', max_iter=-1, probability=False, random_state=None,
  shrinking=True, tol=0.001, verbose=False))])
```

In [12]:

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

Out[12]:

```
0.8599
```

The model did a great job when we used the whole features.

In [13]:

```
pca_svc = Pipeline([
    ('ss', StandardScaler()),
    ('pca', PCA(n_components = 5)),
    ('svc', SVC())
])
```

In [14]:

```
pca_svc.fit(X_train, y_train)
```

Out[14]:

```
Pipeline(memory=None,
     steps=[('ss', StandardScaler(copy=True, with_mean=True, with_std=True)), ('pca', PCA(copy=True, it
erated_power='auto', n_components=5, random_state=None,
  svd_solver='auto', tol=0.0, whiten=False)), ('svc', SVC(C=1.0, cache_size=200, class_weight=None, coe
f0=0.0,
  decision_function_shape='ovr', degree=3, gamma='auto_deprecated',
  kernel='rbf', max_iter=-1, probability=False, random_state=None,
  shrinking=True, tol=0.001, verbose=False))])
```

In [15]:

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

Out[15]:

```
0.797
```

We still need to use all the features we have to get the best accuracy possible.
The SVC model is probably as good as it can get, but I will try to squeeze some more

In [9]:

```
ss_svc = Pipeline([
    ('ss', StandardScaler()),
    ('svm', SVC(probability = True))
])
```

In [17]:

```
# params_ss_svc = {
#     'svm__C' : [1, .5, 0.01],
#     'svm__probability' : [True],
# }
```

In [18]:

```
# gs = GridSearchCV(estimator = ss_svc, param_grid = params_ss_svc, cv = 5)
```

In [19]:

```
# # the kernel kept getting stuck at this cell, so I'll comment it out and go without fine tuning.
# gs.fit(X_train, y_train)
```

In [10]:

```
ss_svc.fit(X_train, y_train)
```

Out[10]:

```
Pipeline(memory=None,
     steps=[('ss', StandardScaler(copy=True, with_mean=True, with_std=True)), ('svm', SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0,
  decision_function_shape='ovr', degree=3, gamma='auto_deprecated',
  kernel='rbf', max_iter=-1, probability=True, random_state=None,
  shrinking=True, tol=0.001, verbose=False))])
```

In [11]:

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

Out[11]:

```
0.9832
```

In [11]:

```
# we have imbalanced classes,
ss_svc = Pipeline([
    ('ss', StandardScaler()),
    ('svc', SVC(class_weight = 'balanced', probability = True))
])
```

In [12]:

```
ss_svc.fit(X_train, y_train)
```

Out[12]:

```
Pipeline(memory=None,
     steps=[('ss', StandardScaler(copy=True, with_mean=True, with_std=True)), ('svc', SVC(C=1.0, cache_size=200, class_weight='balanced', coef0=0.0,
  decision_function_shape='ovr', degree=3, gamma='auto_deprecated',
  kernel='rbf', max_iter=-1, probability=True, random_state=None,
  shrinking=True, tol=0.001, verbose=False))])
```

In [13]:

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

Out[13]:

```
0.9865
```

Just quickly evaluating what I believe to be the best model we can get.

In [14]:

```
# get classes and probabilities predictions
preds = ss_svc.predict(X_test)
preds_probs = ss_svc.predict_proba(X_test)[:,1]
```

In [16]:

```
from sklearn.metrics import roc_auc_score, roc_curve, f1_score
```

In [17]:

```
# get AUC score, and f1-score
print(f"roc_auc score: {roc_auc_score(y_test, preds).round(3)}")
print(f"f1-score: {f1_score(y_test, preds).round(3)}")
```

```
roc_auc score: 0.976
f1-score: 0.966
```

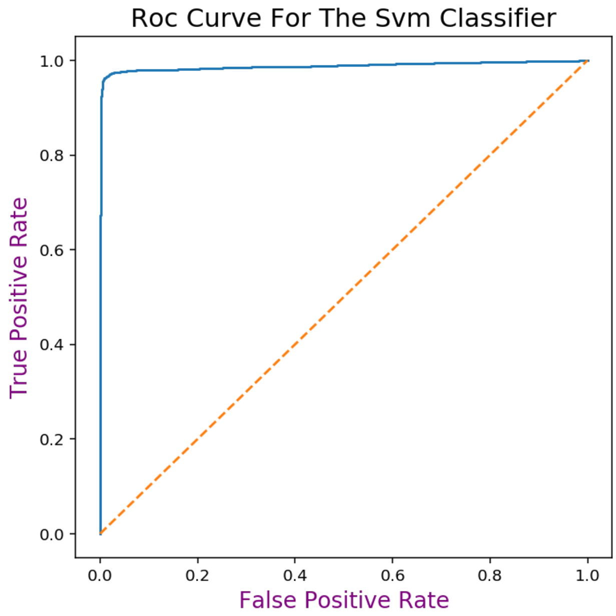
In [19]:

```
def plot_roc(preds_probs):

    """
    Parameters:
    -----------
    preds_probs: model.predict_proba(X_test)[:,1]

    """
    # ROC-curve
    fpr, tpr, _ = roc_curve(y_test, preds_probs)
```

```
    plt.figure(figsize = (6,6))
    plt.plot(fpr, tpr);
    plt.plot([0,max(y_test)],[0, max(y_test)], '--'); # it takes only encoded numerical y
    plt.title('ROC curve for the SVM classifier'.title(), fontsize = 16);
    plt.xlabel('false positive rate'.title(), fontsize = 14, color = 'purple');
    plt.ylabel('true positive rate'.title(), fontsize = 14, color = 'purple');
```

In [20]:

```
plot_roc(preds_probs)
```



In [24]:

```
# saving the model
import pickle
filename = 'model_1_svc.sav'
pickle.dump(ss_svc, open(filename, 'wb'))
```

In [ ]:

```

```

In [1]:

```
import pandas as pd

import matplotlib.pyplot as plt
%config InlineBackend.figure_format = 'retina'

import cleaning_functions as clean
```

In [2]:

```
data = pd.read_csv("Code_challenge_train.csv")
```

In [3]:

```
data = clean.cleaning(data)
```

In [4]:

```
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
```

In [5]:

```
X = data.drop(['y'], axis = 1)
y = data['y']
```

In [6]:

```
X_train, X_test, y_train, y_test = train_test_split(X, y, stratify = y, random_state = 72019)
```

In [7]:

```
ss = StandardScaler()

X_train = ss.fit_transform(X_train)
X_test = ss.transform(X_test)
```

```
/anaconda3/lib/python3.6/site-packages/sklearn/preprocessing/data.py:625: DataConversionWarning: Data w
ith input dtype uint8, int64, float64 were all converted to float64 by StandardScaler.
  return self.partial_fit(X, y)
/anaconda3/lib/python3.6/site-packages/sklearn/base.py:462: DataConversionWarning: Data with input dtyp
e uint8, int64, float64 were all converted to float64 by StandardScaler.
  return self.fit(X, **fit_params).transform(X)
/anaconda3/lib/python3.6/site-packages/ipykernel_launcher.py:4: DataConversionWarning: Data with input
dtype uint8, int64, float64 were all converted to float64 by StandardScaler.
  after removing the cwd from sys.path.
```

In [8]:

```
import warnings
warnings.filterwarnings("ignore")
```

In [9]:

```
from keras.models import Sequential
from keras.layers import Dense
```

```
Using TensorFlow backend.
```

In [10]:

```
# from keras.metrics import binary_accuracy
```

In [11]:

```
def run_model(nodes, epk, btch):

    """
    Parameters:
    -----------
    nodes: Goes in Dense layer, how many nodes you want.
    epk: number of epochs, goes in training the model part
    btch: batch_size, goes in training the model part.
```

```
    """

    model = Sequential()

    # input
    model.add(Dense(nodes, activation = 'relu', input_shape = (X_train.shape[1], )))

    # one hidden layer
    model.add(Dense(nodes, activation = 'relu'))

    # output layer
    model.add(Dense(1, activation = 'sigmoid'))

    # compile
    model.compile(optimizer = 'adam',
                  loss = 'binary_crossentropy',
                  metrics = ['accuracy'])

    history = model.fit(X_train, y_train,
                        epochs = epk,
                        batch_size = btch,
                        validation_data = (X_test, y_test))

    return history
```

In [33]:

```
def plot_loss_acc(history):

    """
    Needs the outcome of run_model() as input.
    Plots accuracy and loss over training and validation
    """

#     loss_values = history.history['loss']
#     val_loss_values = history.history['val_loss']

    epochs = range(1, len(history.history['loss']) + 1)


    fig, ax = plt.subplots(nrows = 1, ncols = 2, figsize = (12,5))
    ax[0].plot(epochs, history.history['loss'], 'bo', label = 'training loss'.title())
    ax[0].plot(epochs, history.history['val_loss'], 'b', label = 'validation loss'.title())
    ax[0].set_title('Training and Validation Loss')
    ax[0].set_xlabel('Epochs')
    ax[0].set_ylabel('Loss')
    ax[0].set_xticks(ticks = epochs)
    ax[0].legend();


    # plotting the values of the accuracy of training and validation
    epochs = range(1, len(history.history['acc']) + 1)

    ax[1].plot(epochs, history.history['acc'], 'bo', label = 'training accuracy'.title())
    ax[1].plot(epochs, history.history['val_acc'], 'b', label = 'validation accuracy'.title())
    ax[1].set_title('Training and Validation Accuracy')
    ax[1].set_xlabel('Epochs')
    ax[1].set_xticks(ticks = epochs)
    ax[1].set_ylabel('Accuracy')
    ax[1].legend();
```

In [46]:

```
# try 16 nodes, 20 epochs, 512 batch_size
# reduce epochs to 10 if overfitting

# try 32 nodes, then variations. we're trying to beat 98.32% we got in SVM
```

In [13]:

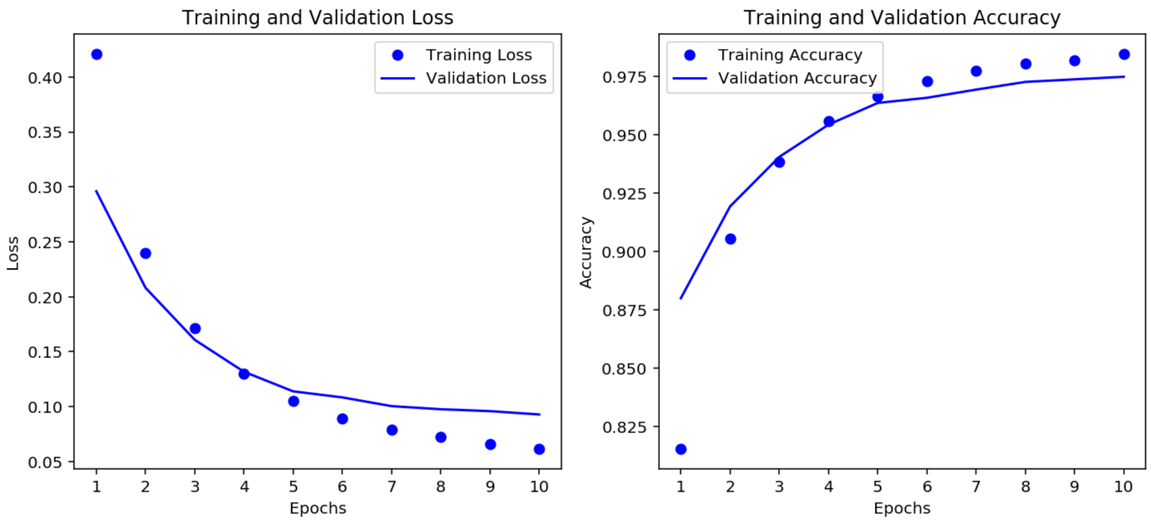
```
hist_1 = run_model(16, 10, 128)
```

```
WARNING:tensorflow:From /anaconda3/lib/python3.6/site-packages/tensorflow/python/framework/op_def_libra
ry.py:263: colocate_with (from tensorflow.python.framework.ops) is deprecated and will be removed in a
future version.
Instructions for updating:
Colocations handled automatically by placer.
```

```
WARNING:tensorflow:From /anaconda3/lib/python3.6/site-packages/tensorflow/python/ops/math_ops.py:3066:
to_int32 (from tensorflow.python.ops.math_ops) is deprecated and will be removed in a future version.
Instructions for updating:
Use tf.cast instead.
Train on 30000 samples, validate on 10000 samples
Epoch 1/10
30000/30000 [==============================] - 1s 48us/step - loss: 0.4212 - acc: 0.8156 - val_loss: 0.
2960 - val_acc: 0.8800
Epoch 2/10
30000/30000 [==============================] - 1s 22us/step - loss: 0.2401 - acc: 0.9054 - val_loss: 0.
2082 - val_acc: 0.9193
Epoch 3/10
30000/30000 [==============================] - 1s 27us/step - loss: 0.1714 - acc: 0.9383 - val_loss: 0.
1608 - val_acc: 0.9405
Epoch 4/10
30000/30000 [==============================] - 1s 21us/step - loss: 0.1302 - acc: 0.9559 - val_loss: 0.
1318 - val_acc: 0.9543
Epoch 5/10
30000/30000 [==============================] - 1s 21us/step - loss: 0.1052 - acc: 0.9664 - val_loss: 0.
1139 - val_acc: 0.9636
Epoch 6/10
30000/30000 [==============================] - 1s 27us/step - loss: 0.0894 - acc: 0.9728 - val_loss: 0.
1084 - val_acc: 0.9658
Epoch 7/10
30000/30000 [==============================] - 1s 21us/step - loss: 0.0789 - acc: 0.9775 - val_loss: 0.
1005 - val_acc: 0.9693
Epoch 8/10
30000/30000 [==============================] - 1s 21us/step - loss: 0.0721 - acc: 0.9805 - val_loss: 0.
0977 - val_acc: 0.9726
Epoch 9/10
30000/30000 [==============================] - 1s 28us/step - loss: 0.0660 - acc: 0.9817 - val_loss: 0.
0959 - val_acc: 0.9737
Epoch 10/10
30000/30000 [==============================] - 1s 21us/step - loss: 0.0617 - acc: 0.9847 - val_loss: 0.
0929 - val_acc: 0.9748
```

In [34]:

```
plot_loss_acc(hist_1)
```



In [35]:

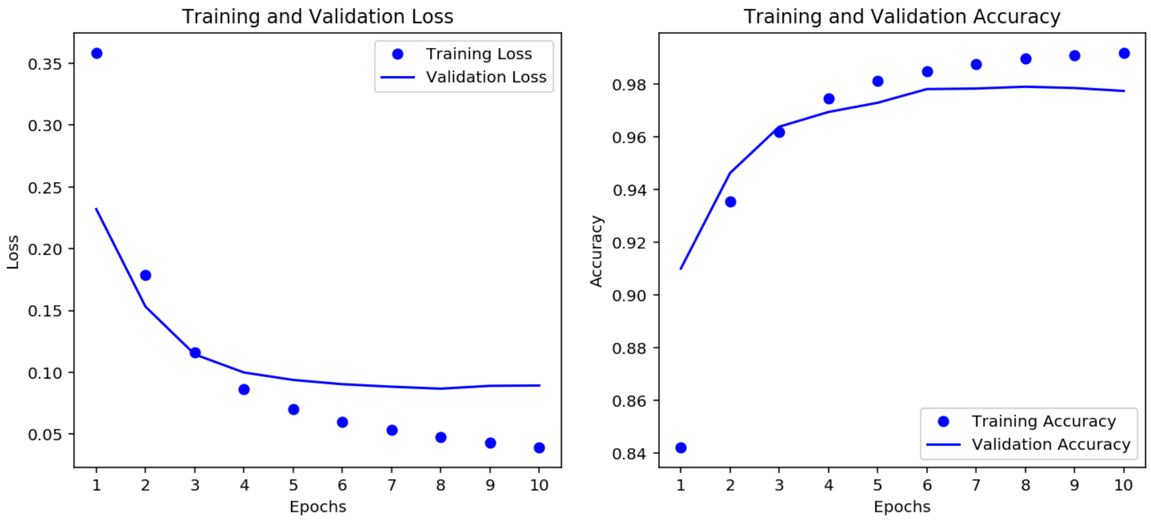
```
hist_2 = run_model(32, 10, 128)
```

```
Train on 30000 samples, validate on 10000 samples
Epoch 1/10
30000/30000 [==============================] - 2s 56us/step - loss: 0.3584 - acc: 0.8421 - val_loss: 0.
2319 - val_acc: 0.9100
Epoch 2/10
30000/30000 [==============================] - 1s 24us/step - loss: 0.1786 - acc: 0.9355 - val_loss: 0.
1531 - val_acc: 0.9463
Epoch 3/10
30000/30000 [==============================] - 1s 30us/step - loss: 0.1163 - acc: 0.9618 - val_loss: 0.
1145 - val_acc: 0.9639
Epoch 4/10
30000/30000 [==============================] - 1s 24us/step - loss: 0.0862 - acc: 0.9747 - val_loss: 0.
0999 - val_acc: 0.9695
```

```
Epoch 5/10
30000/30000 [==============================] - 1s 24us/step - loss: 0.0703 - acc: 0.9814 - val_loss: 0.
0939 - val_acc: 0.9730
Epoch 6/10
30000/30000 [==============================] - 1s 31us/step - loss: 0.0602 - acc: 0.9850 - val_loss: 0.
0904 - val_acc: 0.9782
Epoch 7/10
30000/30000 [==============================] - 1s 24us/step - loss: 0.0533 - acc: 0.9876 - val_loss: 0.
0884 - val_acc: 0.9784
Epoch 8/10
30000/30000 [==============================] - 1s 24us/step - loss: 0.0477 - acc: 0.9899 - val_loss: 0.
0868 - val_acc: 0.9791
Epoch 9/10
30000/30000 [==============================] - 1s 26us/step - loss: 0.0428 - acc: 0.9910 - val_loss: 0.
0891 - val_acc: 0.9786
Epoch 10/10
30000/30000 [==============================] - 1s 25us/step - loss: 0.0394 - acc: 0.9920 - val_loss: 0.
0894 - val_acc: 0.9775
```

In [36]:

```
plot_loss_acc(hist_2)
```



It's overfitting worse than the model before. Tweaking some more below

In [37]:

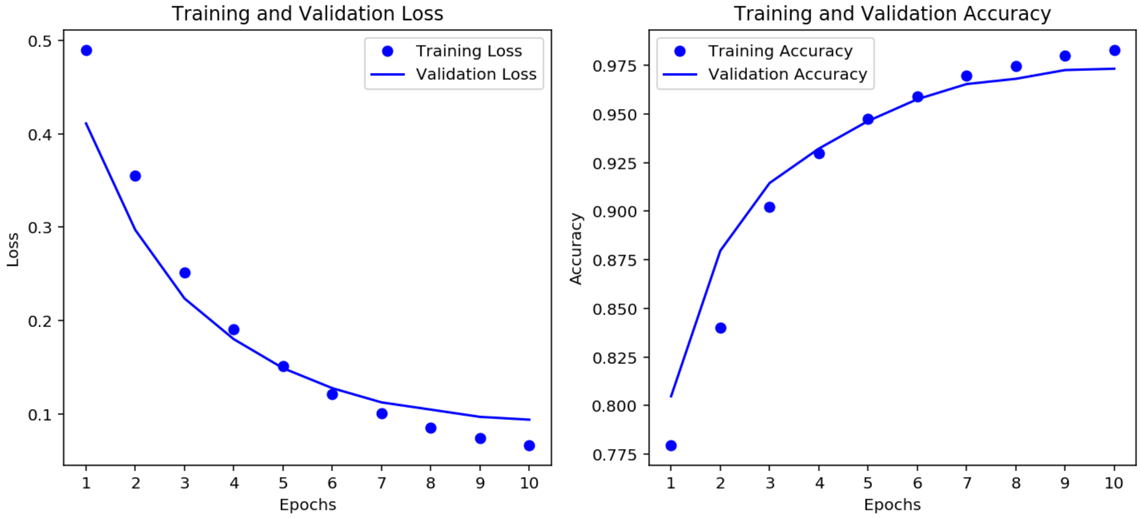
```
hist_3 = run_model(32, 10, 512)
```

```
Train on 30000 samples, validate on 10000 samples
Epoch 1/10
30000/30000 [==============================] - 1s 40us/step - loss: 0.4898 - acc: 0.7797 - val_loss: 0.
4110 - val_acc: 0.8048
Epoch 2/10
30000/30000 [==============================] - 0s 9us/step - loss: 0.3556 - acc: 0.8403 - val_loss: 0.2
970 - val_acc: 0.8797
Epoch 3/10
30000/30000 [==============================] - 0s 9us/step - loss: 0.2514 - acc: 0.9022 - val_loss: 0.2
237 - val_acc: 0.9145
Epoch 4/10
30000/30000 [==============================] - 0s 15us/step - loss: 0.1911 - acc: 0.9297 - val_loss: 0.
1802 - val_acc: 0.9322
Epoch 5/10
30000/30000 [==============================] - 0s 9us/step - loss: 0.1512 - acc: 0.9473 - val_loss: 0.1
492 - val_acc: 0.9464
Epoch 6/10
30000/30000 [==============================] - 0s 9us/step - loss: 0.1217 - acc: 0.9591 - val_loss: 0.1
278 - val_acc: 0.9576
Epoch 7/10
30000/30000 [==============================] - 0s 9us/step - loss: 0.1006 - acc: 0.9696 - val_loss: 0.1
126 - val_acc: 0.9654
Epoch 8/10
30000/30000 [==============================] - 0s 9us/step - loss: 0.0855 - acc: 0.9745 - val_loss: 0.1
048 - val_acc: 0.9681
Epoch 9/10
30000/30000 [==============================] - 0s 9us/step - loss: 0.0748 - acc: 0.9799 - val_loss: 0.0
971 - val_acc: 0.9726
```

Epoch 10/10
30000/30000 [==============================] - 0s 9us/step - loss: 0.0668 - acc: 0.9830 - val_loss: 0.0941 - val_acc: 0.9733

In [38]:

```
plot_loss_acc(hist_3)
```



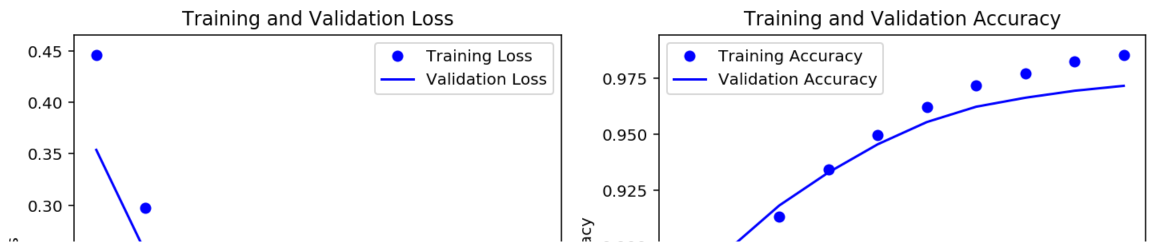
In [39]:

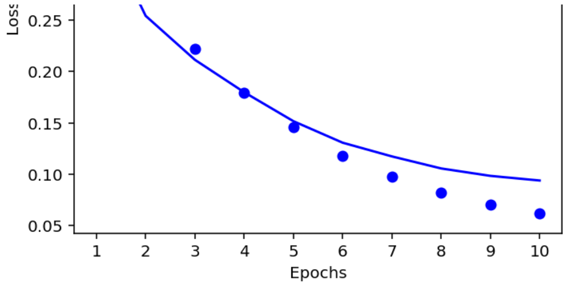
```
hist_4 = run_model(64, 10, 1024)
```

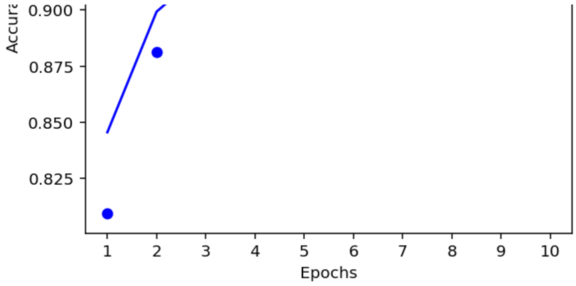
```
Train on 30000 samples, validate on 10000 samples
Epoch 1/10
30000/30000 [==============================] - 1s 42us/step - loss: 0.4462 - acc: 0.8095 - val_loss: 0.3538 - val_acc: 0.8456
Epoch 2/10
30000/30000 [==============================] - 0s 8us/step - loss: 0.2974 - acc: 0.8812 - val_loss: 0.2543 - val_acc: 0.8993
Epoch 3/10
30000/30000 [==============================] - 0s 14us/step - loss: 0.2225 - acc: 0.9132 - val_loss: 0.2115 - val_acc: 0.9183
Epoch 4/10
30000/30000 [==============================] - 0s 8us/step - loss: 0.1797 - acc: 0.9343 - val_loss: 0.1799 - val_acc: 0.9330
Epoch 5/10
30000/30000 [==============================] - 0s 8us/step - loss: 0.1460 - acc: 0.9496 - val_loss: 0.1518 - val_acc: 0.9455
Epoch 6/10
30000/30000 [==============================] - 0s 8us/step - loss: 0.1181 - acc: 0.9622 - val_loss: 0.1308 - val_acc: 0.9554
Epoch 7/10
30000/30000 [==============================] - 0s 8us/step - loss: 0.0978 - acc: 0.9717 - val_loss: 0.1175 - val_acc: 0.9622
Epoch 8/10
30000/30000 [==============================] - 0s 8us/step - loss: 0.0819 - acc: 0.9769 - val_loss: 0.1057 - val_acc: 0.9662
Epoch 9/10
30000/30000 [==============================] - 0s 8us/step - loss: 0.0705 - acc: 0.9823 - val_loss: 0.0985 - val_acc: 0.9693
Epoch 10/10
30000/30000 [==============================] - 0s 8us/step - loss: 0.0621 - acc: 0.9853 - val_loss: 0.0940 - val_acc: 0.9715
```

In [40]:

```
plot_loss_acc(hist_4)
```







In [44]:

```
def run_model_2hidden(nodes, epk, btch):

    """
    Parameters:
    -----------
    nodes: Goes in Dense layer, how many nodes you want.
    epk: number of epochs, goes in training the model part
    btch: batch_size, goes in training the model part.
    """

    model = Sequential()

    # input
    model.add(Dense(nodes, activation = 'relu', input_shape = (X_train.shape[1], )))

    # one hidden layer
    model.add(Dense(nodes, activation = 'relu'))

    # second hidden layer
    model.add(Dense(nodes, activation = 'relu'))

    # output layer
    model.add(Dense(1, activation = 'sigmoid'))

    # compile
    model.compile(optimizer = 'adam',
                  loss = 'binary_crossentropy',
                  metrics = ['accuracy'])

    history = model.fit(X_train, y_train,
                        epochs = epk,
                        batch_size = btch,
                        validation_data = (X_test, y_test))

    return history
```

In [45]:

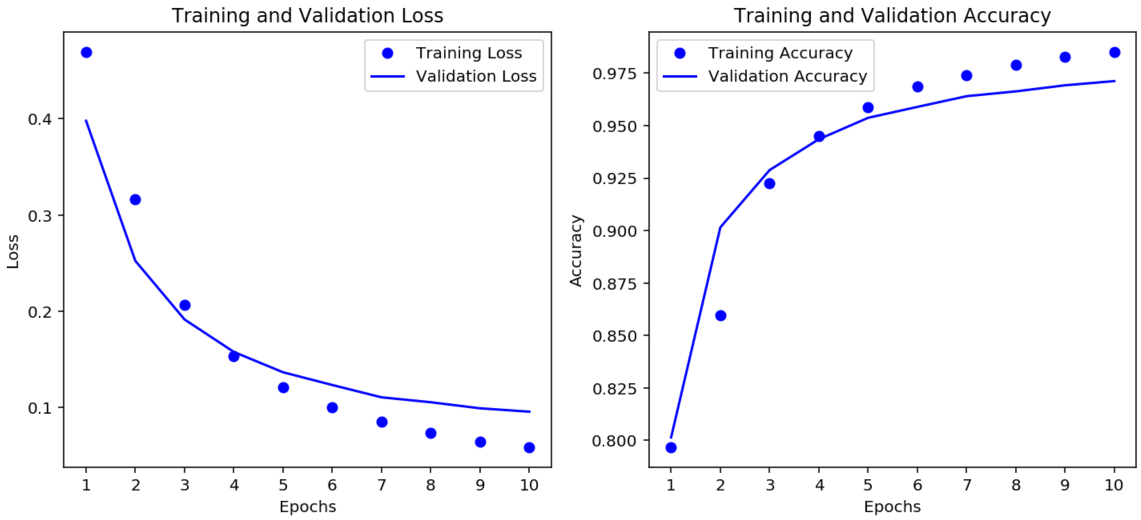
```
hist_5 = run_model_2hidden(32, 10, 512)
```

```
Train on 30000 samples, validate on 10000 samples
Epoch 1/10
30000/30000 [==============================] - 2s 50us/step - loss: 0.4696 - acc: 0.7968 - val_loss: 0.
3980 - val_acc: 0.8014
Epoch 2/10
30000/30000 [==============================] - 0s 16us/step - loss: 0.3165 - acc: 0.8597 - val_loss: 0.
2525 - val_acc: 0.9015
Epoch 3/10
30000/30000 [==============================] - 0s 10us/step - loss: 0.2065 - acc: 0.9225 - val_loss: 0.
1915 - val_acc: 0.9288
Epoch 4/10
30000/30000 [==============================] - 0s 10us/step - loss: 0.1539 - acc: 0.9452 - val_loss: 0.
1578 - val_acc: 0.9435
Epoch 5/10
30000/30000 [==============================] - 0s 10us/step - loss: 0.1214 - acc: 0.9586 - val_loss: 0.
1367 - val_acc: 0.9537
Epoch 6/10
30000/30000 [==============================] - 0s 10us/step - loss: 0.1000 - acc: 0.9688 - val_loss: 0.
1235 - val_acc: 0.9589
Epoch 7/10
30000/30000 [==============================] - 0s 10us/step - loss: 0.0855 - acc: 0.9740 - val_loss: 0.
1107 - val_acc: 0.9640
Epoch 8/10
30000/30000 [==============================] - 0s 17us/step - loss: 0.0737 - acc: 0.9789 - val_loss: 0.
```

```
1056 - val_acc: 0.9663
Epoch 9/10
30000/30000 [==============================] - 0s 10us/step - loss: 0.0648 - acc: 0.9829 - val_loss: 0.
0993 - val_acc: 0.9692
Epoch 10/10
30000/30000 [==============================] - 0s 10us/step - loss: 0.0589 - acc: 0.9852 - val_loss: 0.
0957 - val_acc: 0.9712
```

In [46]:

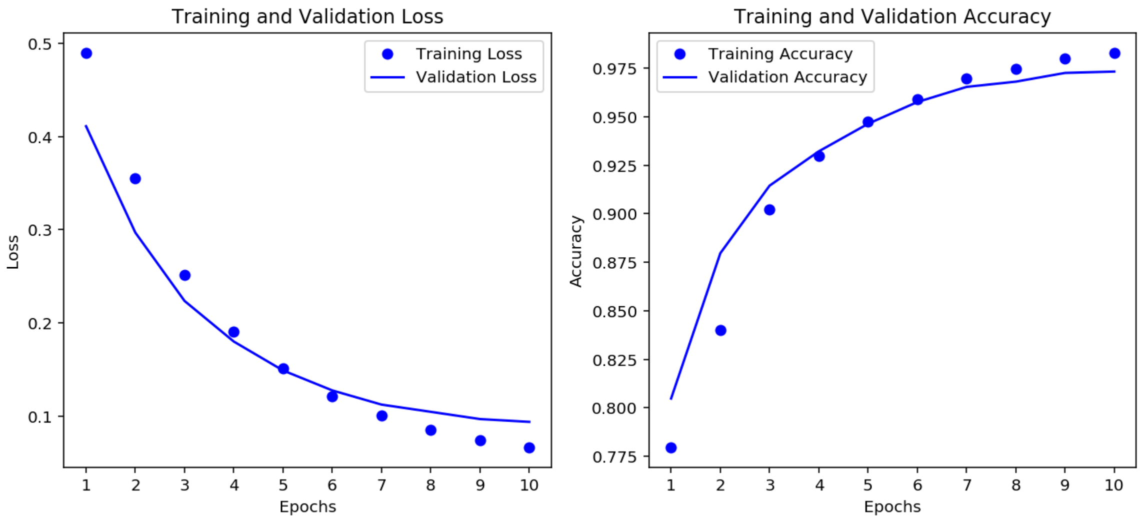
```
plot_loss_acc(hist_5)
```



It's doing worse than with one hidden layer, it's overfitting. Compare to the best model we had so far:

In [47]:

```
plot_loss_acc(hist_3)
```



Best accuracy on validation set, was in hist_3 with 97.33% It's is overfitting a little after the 6th epoch. I'll make the cut at the 6th epoch, and see how much worse the accuracy would be

In [48]:

```
hist_3_2 = run_model(32, 6, 512)
```

```
Train on 30000 samples, validate on 10000 samples
Epoch 1/6
30000/30000 [==============================] - 2s 51us/step - loss: 0.5896 - acc: 0.6828 - val_loss: 0.
4281 - val_acc: 0.8057
Epoch 2/6
30000/30000 [==============================] - 0s 9us/step - loss: 0.3663 - acc: 0.8352 - val_loss: 0.3
204 - val_acc: 0.8658
Epoch 3/6
30000/30000 [==============================] - 0s 9us/step - loss: 0.2737 - acc: 0.8926 - val_loss: 0.2
501 - val_acc: 0.9029
```

```
Epoch 4/6
30000/30000 [==============================] - 0s 9us/step - loss: 0.2157 - acc: 0.9192 - val_loss: 0.2
083 - val_acc: 0.9210
Epoch 5/6
30000/30000 [==============================] - 0s 9us/step - loss: 0.1767 - acc: 0.9376 - val_loss: 0.1
769 - val_acc: 0.9352
Epoch 6/6
30000/30000 [==============================] - 0s 9us/step - loss: 0.1460 - acc: 0.9509 - val_loss: 0.1
513 - val_acc: 0.9469
```

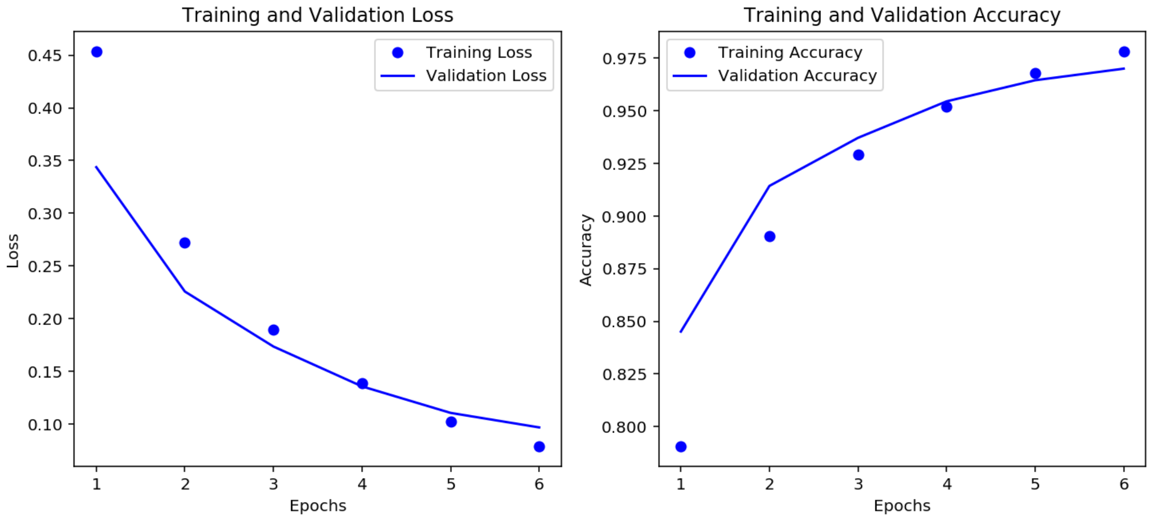
In [49]:

```
hist_3_3 = run_model(64, 6, 512)
```

```
Train on 30000 samples, validate on 10000 samples
Epoch 1/6
30000/30000 [==============================] - 2s 50us/step - loss: 0.4537 - acc: 0.7907 - val_loss: 0.
3437 - val_acc: 0.8451
Epoch 2/6
30000/30000 [==============================] - 0s 11us/step - loss: 0.2723 - acc: 0.8903 - val_loss: 0.
2258 - val_acc: 0.9142
Epoch 3/6
30000/30000 [==============================] - 0s 11us/step - loss: 0.1894 - acc: 0.9289 - val_loss: 0.
1736 - val_acc: 0.9370
Epoch 4/6
30000/30000 [==============================] - 1s 17us/step - loss: 0.1386 - acc: 0.9517 - val_loss: 0.
1358 - val_acc: 0.9543
Epoch 5/6
30000/30000 [==============================] - 0s 11us/step - loss: 0.1022 - acc: 0.9679 - val_loss: 0.
1107 - val_acc: 0.9643
Epoch 6/6
30000/30000 [==============================] - 0s 11us/step - loss: 0.0790 - acc: 0.9781 - val_loss: 0.
0969 - val_acc: 0.9698
```

In [50]:

```
plot_loss_acc(hist_3_3)
```



Running the model independently, in order to save it

In [63]:

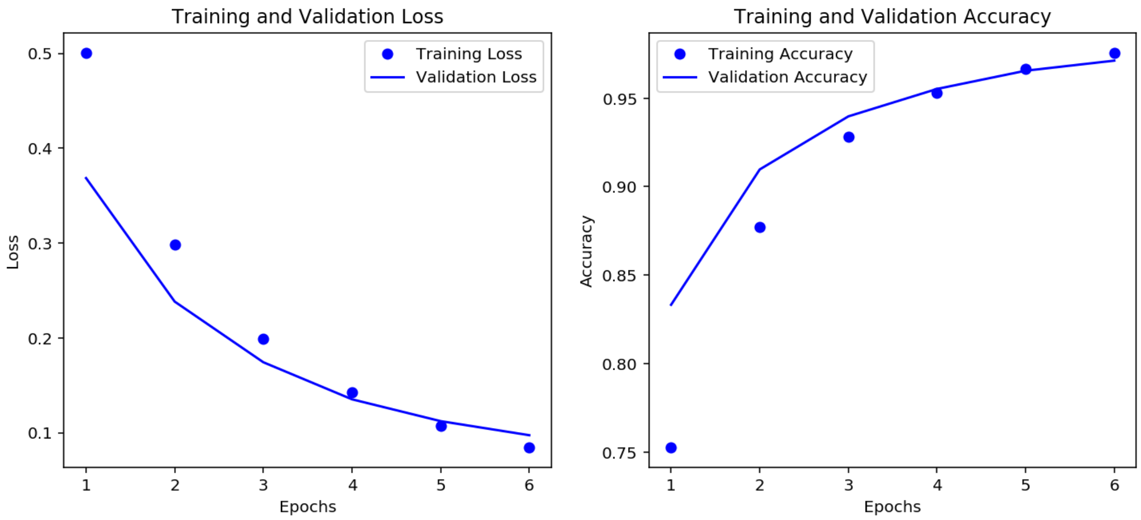
```
# hist_3_3 = run_model(64, 6, 512)
model = Sequential()

# input
model.add(Dense(64, activation = 'relu', input_shape = (X_train.shape[1], )))

# one hidden layer
model.add(Dense(64, activation = 'relu'))

# output layer
model.add(Dense(1, activation = 'sigmoid'))
```

```python
# compile
model.compile(optimizer = 'adam',
              loss = 'binary_crossentropy',
              metrics = ['accuracy'])

history = model.fit(X_train, y_train,
                epochs = 6,
                batch_size = 512,
                validation_data = (X_test, y_test))
```

```
Train on 30000 samples, validate on 10000 samples
Epoch 1/6
30000/30000 [==============================] - 2s 52us/step - loss: 0.5007 - acc: 0.7526 - val_loss: 0.
3685 - val_acc: 0.8332
Epoch 2/6
30000/30000 [==============================] - 0s 11us/step - loss: 0.2986 - acc: 0.8770 - val_loss: 0.
2383 - val_acc: 0.9097
Epoch 3/6
30000/30000 [==============================] - 0s 11us/step - loss: 0.1992 - acc: 0.9282 - val_loss: 0.
1743 - val_acc: 0.9397
Epoch 4/6
30000/30000 [==============================] - 1s 17us/step - loss: 0.1425 - acc: 0.9529 - val_loss: 0.
1352 - val_acc: 0.9552
Epoch 5/6
30000/30000 [==============================] - 0s 11us/step - loss: 0.1070 - acc: 0.9667 - val_loss: 0.
1122 - val_acc: 0.9655
Epoch 6/6
30000/30000 [==============================] - 0s 11us/step - loss: 0.0846 - acc: 0.9757 - val_loss: 0.
0974 - val_acc: 0.9712
```

In [64]:

```python
plot_loss_acc(history)
```



In [ ]:

```python

```

In [65]:

```python
# saving it for later use
import pickle
filename = 'model_2_nn.sav'
pickle.dump(model, open(filename, 'wb'))
```

In [ ]:

```python

```

In [1]:

```
import pandas as pd

import matplotlib.pyplot as plt
%config InlineBackend.figure_format = 'retina'

import cleaning_functions as clean
```

In [2]:

```
data = pd.read_csv('Code_challenge_train.csv')
```

In [3]:

```
data = clean.cleaning(data)
```

In [4]:

```
nulls = data.isnull().sum().to_frame()
nulls.loc[nulls[0] != 0, :]
```

Out[4]:

0

In [5]:

```
from sklearn.model_selection import train_test_split

X = data.drop('y', axis = 1)
y = data['y']

X_train, X_test, y_train, y_test = train_test_split(X, y, stratify = y, random_state = 72019)
```

In [6]:

```
import warnings
warnings.filterwarnings('ignore')
```

In [7]:

```
from xgboost import XGBClassifier

from sklearn.pipeline import Pipeline
from sklearn.model_selection import RandomizedSearchCV, GridSearchCV
from sklearn.preprocessing import StandardScaler
```

In [8]:

```
ss_xg = Pipeline([
    ('ss', StandardScaler()),
    ('xg', XGBClassifier())
])
```

In [9]:

```
ss_xg.fit(X_train, y_train)
```

Out[9]:

```
Pipeline(memory=None,
     steps=[('ss', StandardScaler(copy=True, with_mean=True, with_std=True)), ('xg', XGBClassifier(base
_score=0.5, booster='gbtree', colsample_bylevel=1,
       colsample_bynode=1, colsample_bytree=1, gamma=0, learning_rate=0.1,
       max_delta_step=0, max_depth=3, min_child_weight=1, missing=None,
    ...
       reg_lambda=1, scale_pos_weight=1, seed=None, silent=None,
       subsample=1, verbosity=1))])
```

In [11]:

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

Out[11]:

```
0.9041
```

In [15]:

```
pipe_params = {
    'xg__n_estimators' : [100, 50, 114],
    'xg__max_depth' : [3, 1, 5],
    'xg__learning_rate' : [.1, .5],
    'xg__reg_alpha' : [0,.3]
}
```

In [16]:

```
gs = GridSearchCV(estimator = ss_xg, param_grid = pipe_params, cv = 5)
```

In [17]:

```
gs.fit(X_train, y_train)
```

Out[17]:

```
GridSearchCV(cv=5, error_score='raise-deprecating',
       estimator=Pipeline(memory=None,
     steps=[('ss', StandardScaler(copy=True, with_mean=True, with_std=True)), ('xg', XGBClassifier(base
_score=0.5, booster='gbtree', colsample_bylevel=1,
       colsample_bynode=1, colsample_bytree=1, gamma=0, learning_rate=0.1,
       max_delta_step=0, max_depth=3, min_child_weight=1, missing=None,
    ...
       reg_lambda=1, scale_pos_weight=1, seed=None, silent=None,
       subsample=1, verbosity=1))]),
       fit_params=None, iid='warn', n_jobs=None,
       param_grid={'xg__n_estimators': [100, 50, 114], 'xg__max_depth': [3, 1, 5], 'xg__learning_rate':
[0.1, 0.5], 'xg__reg_alpha': [0, 0.3]},
       pre_dispatch='2*n_jobs', refit=True, return_train_score='warn',
       scoring=None, verbose=0)
```

In [19]:

```
gs.best_params_
```

Out[19]:

```
{'xg__learning_rate': 0.5,
 'xg__max_depth': 5,
 'xg__n_estimators': 114,
 'xg__reg_alpha': 0.3}
```

In [20]:

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

Out[20]:

```
0.97
```

In [21]:

```
# saving the model
import pickle
filename = 'model_2_xgboost.sav'
pickle.dump(gs, open(filename, 'wb'))
```

In [ ]:

```

```

In [1]:

```
import pickle

import pandas as pd

import matplotlib.pyplot as plt
%config InlineBackend.figure_format = 'retina'

import cleaning_functions as clean
```

In [2]:

```
train_df = pd.read_csv('Code_challenge_train.csv')
test_df = pd.read_csv('Code_challenge_test.csv')
```

In [3]:

```
train_df = clean.cleaning(train_df)
```

In [4]:

```
test_df = clean.cleaning(test_df)
```

In [6]:

```
X = train_df.drop(['y'], axis = 1)
y = train_df['y']
```

predicting from the first saved model. it was Support Victor Machine Classifier

In [7]:

```
# load the saved SVM model
model_1_svc = pickle.load(open('model_1_svc.sav', 'rb'))
```

In [8]:

```
# predicting on the hold-out set
results_1 = model_1_svc.predict_proba(test_df)[:,1]
print(results_1)
```

```
/anaconda3/lib/python3.6/site-packages/sklearn/pipeline.py:381: DataConversionWarning: Data with input
dtype uint8, int64, float64 were all converted to float64 by StandardScaler.
  Xt = transform.transform(Xt)
```

```
[1.53137415e-02 2.89409608e-04 4.66431509e-03 ... 1.23553230e-04
 9.11889728e-01 9.79952738e-01]
```

In [10]:

```
# convert to data frame
results_1 = pd.DataFrame(results_1, columns = ['1'])
```

In [13]:

```
# saving predictions
results_1.to_csv('results1.csv', header = '1', index = False)
```

Scoring and evaluating the first model: SVM

In [26]:

```
from sklearn.metrics import roc_auc_score, roc_curve, confusion_matrix, f1_score
```

In [27]:

```
import warnings
warnings.filterwarnings('ignore')
```

In [28]:

```
# get classes and probabilities predictions for the validation set, so to score
```

```
# get classes and probabilities predictions for the validation set, so to score
preds_1 = model_1_svc.predict(X_test)
preds_1_probs = model_1_svc.predict_proba(X_test)[:,1]
```

In [29]:

```
# just so we have the counts of False Positives and False Negatives, to those true ones
# for the first model
cm_1 = pd.DataFrame(confusion_matrix(y_test, preds_1), columns = ['Predicted 0', 'Predicted 1'],
            index = ['Actual 0', 'Actual 1'])
cm_1
```

Out[29]:

	Predicted 0	Predicted 1
Actual 0	7922	48
Actual 1	87	1943

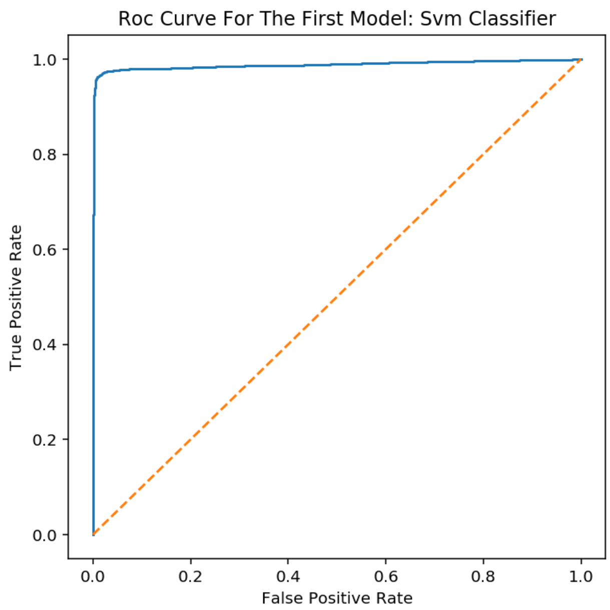
In [30]:

```
# scores
print(f"AUC score: {roc_auc_score(y_test, preds_1).round(3)}")
print(f"f1-score: {f1_score(y_test, preds_1).round(3)}")
```

```
AUC score: 0.976
f1-score: 0.966
```

In [31]:

```
# roc-curve for the first model to visualize
fpr, tpr, _ = roc_curve(y_test, preds_1_probs)

plt.figure(figsize = (6,6))
plt.plot(fpr, tpr);
plt.plot([0,max(y_test)],[0, max(y_test)], '--'); # it takes only encoded numerical y
plt.title('ROC curve for the first model: SVM classifier'.title());
plt.xlabel('false positive rate'.title());
plt.ylabel('true positive rate'.title());
```



predicting from the second saved model. It was a neural network

In [11]:

```
import pickle
model_2_nn = pickle.load(open('model_2_nn.sav', 'rb'))
```

In [31]:

```
results_2 = model_2_nn.predict_proba(test_df, batch_size=512)
```

In [38]:

```
pd.DataFrame(results_2).sample(10)
```

Out[38]:

	0
4316	0.155588
3028	1.000000
6548	0.000000
1971	1.000000
1685	1.000000
3123	1.000000
8397	0.992411
1759	1.000000
1205	0.000000
5305	0.000000

In [41]:

```
results_2 = pd.DataFrame(results_2, columns = ['1'])
```

In [43]:

```
results_2.to_csv('results2.csv', header = '1', index = False)
```

Scoring and evaluating neural networks model

In [51]:

```
results_2 = pd.read_csv('results2.csv')
```

In [23]:

```
from sklearn.model_selection import train_test_split

X = train_df.drop('y', axis = 1)
y = train_df['y']

X_train, X_test, y_train, y_test = train_test_split(X, y, stratify = y, random_state = 72019)
```

In [57]:

```
preds_2 = model_2_nn.predict_classes(X_test)
```

```
/anaconda3/lib/python3.6/site-packages/sklearn/pipeline.py:331: DataConversionWarning: Data with input
dtype uint8, int64, float64 were all converted to float64 by StandardScaler.
  Xt = transform.transform(Xt)
```

In [76]:

```
preds_2_probs = model_2_nn.predict_proba(X_test)
```

```
/anaconda3/lib/python3.6/site-packages/sklearn/pipeline.py:381: DataConversionWarning: Data with input
dtype uint8, int64, float64 were all converted to float64 by StandardScaler.
  Xt = transform.transform(Xt)
```

In [63]:

```
# for the second model
cm_2 = pd.DataFrame(confusion_matrix(y_test, preds_2), columns = ['predicted 0', 'predicted 1'],
            index = ['Actual 0', 'Actual 1'])
cm_2
```

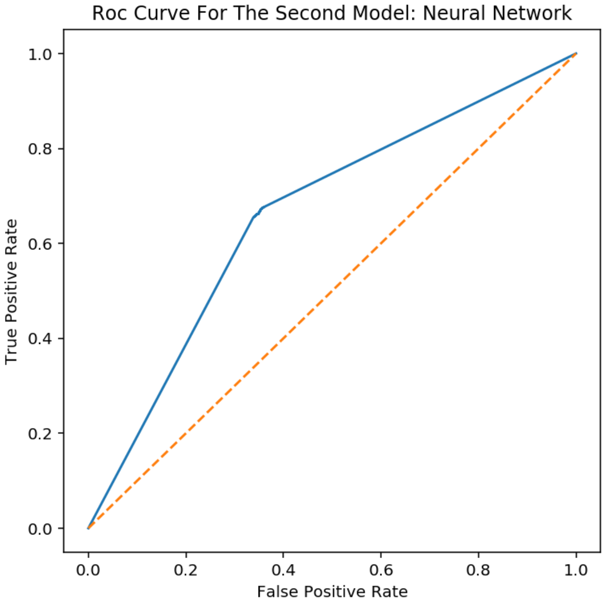
Out[63]:

	predicted 0	predicted 1
Actual 0	5180	2781

	predicted 0	predicted 1
Actual 0	5189	2781
Actual 1	686	1344

In [92]:

```
# roc-curve and auc score for the second model
fpr, tpr, _ = roc_curve(y_test, preds_2_probs)

plt.figure(figsize = (6,6))
plt.plot(fpr, tpr);
plt.plot([0,max(y_test)],[0, max(y_test)], '--'); # it takes only encoded numerical y
plt.title('ROC curve for the second model: Neural network'.title());
plt.xlabel('false positive rate'.title());
plt.ylabel('true positive rate'.title());
```



In [89]:

```
# auc scores
# for the first model
print(f"SVM: {roc_auc_score(y_test, preds_1).round(3)}")

# and the second
print(f"NN: {roc_auc_score(y_test, preds_2).round(3)}")
```

```
SVM: 0.963
NN: 0.657
```

In [91]:

```
# f-1 scores
print(f"SVM: {f1_score(y_test, preds_1).round(3)}")
print(f"NN: {f1_score(y_test, preds_2).round(3)}")
```

```
SVM: 0.957
NN: 0.437
```

An epic failure for the neural network model. I will go back to the drawing board, and focus on the other machine learning model that did second best to SVC

In [93]:

```
%ls
```

```
Code Challenge Instructions.docx*  cleaning_functions.py
Code_challenge_test.csv*           initial_building_up.ipynb
Code_challenge_train.csv*          model_1_svc.sav
EDA.ipynb                          model_2_nn.sav
Modeling_part_1.ipynb              predicting_scoring.ipynb
Modeling_part_2.ipynb              results1.csv
__pycache__/                       results2.csv
```

In [94]:

```
%rm results2.csv
%rm model_2_nn.sav
```

Repeating the process for the XGBoost model

In [33]:

```
model_2_xgboost = pickle.load(open('model_2_xgboost.sav', 'rb'))
results_2 = model_2_xgboost.predict_proba(test_df)[:,1]
```

In [34]:

```
# evaluating the xgboost model
preds_2 = model_2_xgboost.predict(X_test)
preds_2_probs = model_2_xgboost.predict_proba(X_test)
```

In [35]:

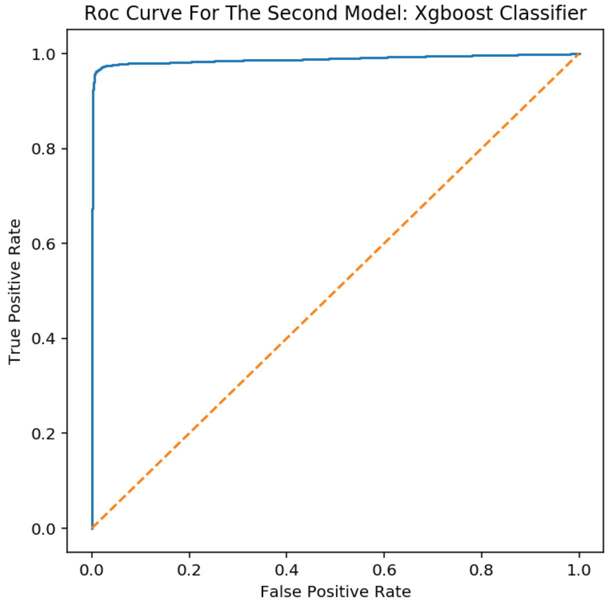
```
cm_2 = pd.DataFrame(confusion_matrix(y_test, preds_2), columns = ['Predicted 0', 'Predicted 1'],
            index = ['Actual 0', 'Actual 1'])
cm_2
```

Out[35]:

	Predicted 0	Predicted 1
Actual 0	7910	60
Actual 1	240	1790

In [36]:

```
# roc-curve and auc score for the first model
fpr, tpr, _ = roc_curve(y_test, preds_1_probs)

plt.figure(figsize = (6,6))
plt.plot(fpr, tpr);
plt.plot([0,max(y_test)],[0, max(y_test)], '--'); # it takes only encoded numerical y
plt.title('ROC curve for the second model: XGBoost classifier'.title());
plt.xlabel('false positive rate'.title());
plt.ylabel('true positive rate'.title());
```



In [37]:

```
# auc scores comparison
# for the first model
print(f"SVM: {roc_auc_score(y_test, preds_1).round(3)}")

# and the second
print(f"XGBoost: {roc_auc_score(y_test, preds_2).round(3)}")
```

```
SVM: 0.976
XGBoost: 0.937
```

In [38]:

```
# f-1 scores comparison
print(f"SVM: {f1_score(y_test, preds_1).round(3)}")
print(f"XGBoost: {f1_score(y_test, preds_2).round(3)}")
```

```
SVM: 0.966
XGBoost: 0.923
```

In [114]:

```
# saving the results of the XGBoost model
pd.DataFrame(results_2, columns = ['1']).to_csv('results2.csv', header = '1', index = False)
```

In [120]:

```
pd.read_csv('results2.csv').sample(10)
```

Out[120]:

	1
5829	0.000923
462	0.996085
2714	0.000657
2152	0.265746
9681	0.035444
1647	0.000339
8232	0.026412
9010	0.002754
965	0.001799
304	0.999118

In [ ]:

```

```