Assignment1 Report

The prediction task is to determine whether a person makes over 50K a year. In this report, I will go through my codes and explain them.

First, I import some libraries and our dataset.

In [3]:

```
# Import some libraries
import pandas as pd
import numpy as np
import matplotlib as mlt
import matplotlib.pyplot as plt
import seaborn as sns
```

In [4]:

```
# Import datasets
train_data=pd.read_csv('/Users/Stylewsxcde991/Desktop/物聯網下商管統計分析/qbs-
competition-1/data/train.csv',index_col=0)
X_test=pd.read_csv('/Users/Stylewsxcde991/Desktop/物聯網下商管統計分析/qbs-
competition-1/data/test.csv',index_col=0)
```

Now let's look some basic information of our dataset.

In [5]:

```
# Some basic information of training data
print('The shape of training data: ' + str(train_data.shape))
print('')
print('The shape of training data: ' + str(X_test.shape))
print('')
print('Basic information of our training data: ')
print(train_data.info())
print('')
print('Basic information of our testing data: ')
print(X_test.info())
The shape of training data: (29514, 15)

The shape of training data: (19328, 14)

Basic information of our training data:
<class 'pandas.core.frame.DataFrame'>
Int64Index: 29514 entries, 2 to 48841
Data columns (total 15 columns):
 #   Column          Non-Null Count  Dtype
```

```
---  ------          --------------  -----
 0   Age             29514 non-null  int64
 1   Workclass       27665 non-null  object
 2   fnlwgt          29514 non-null  int64
 3   Education       29514 non-null  object
 4   Education_Num   29514 non-null  int64
 5   Martial_Status  29514 non-null  object
 6   Occupation      27657 non-null  object
 7   Relationship    29514 non-null  object
 8   Race            29514 non-null  object
 9   Sex             29514 non-null  object
 10  Capital_Gain    29514 non-null  int64
 11  Capital_Loss    29514 non-null  int64
 12  Hours_per_week  29514 non-null  int64
 13  Country         28988 non-null  object
 14  Target          29514 non-null  int64
dtypes: int64(7), object(8)
memory usage: 3.6+ MB
None

Basic information of our testing data:
<class 'pandas.core.frame.DataFrame'>
Int64Index: 19328 entries, 1 to 48842
Data columns (total 14 columns):
 #   Column          Non-Null Count  Dtype
---  ------          --------------  -----
 0   Age             19328 non-null  int64
 1   Workclass       18378 non-null  object
 2   fnlwgt          19328 non-null  int64
 3   Education       19328 non-null  object
 4   Education_Num   19328 non-null  int64
 5   Martial_Status  19328 non-null  object
 6   Occupation      18376 non-null  object
 7   Relationship    19328 non-null  object
 8   Race            19328 non-null  object
 9   Sex             19328 non-null  object
 10  Capital_Gain    19328 non-null  int64
 11  Capital_Loss    19328 non-null  int64
 12  Hours_per_week  19328 non-null  int64
 13  Country         18997 non-null  object
dtypes: int64(6), object(8)
memory usage: 2.2+ MB
None
```

According to above information, the shape of training data is (29514, 15) and the shape of test data is (19328, 14).

Furthermore, notice that we have missing data problem in our training dataset and test dataset (there are Null value in some features).

In particular, we have to deal with the **missing data problem** of 'Workclass', 'Occupation', 'Country' in our training dataset and testing dataset. We can deal with this problem by replacing all Null value with 'unknown'.

In [6]:

```
# Deal with missing data
train_data.Workclass=train_data.Workclass.fillna('unknown')
train_data.Occupation=train_data.Occupation.fillna('unknown')
train_data.Country=train_data.Country.fillna('unknown')
X_test.Workclass=X_test.Workclass.fillna('unknown')
X_test.Occupation=X_test.Occupation.fillna('unknown')
X_test.Country=X_test.Country.fillna('unknown')
```

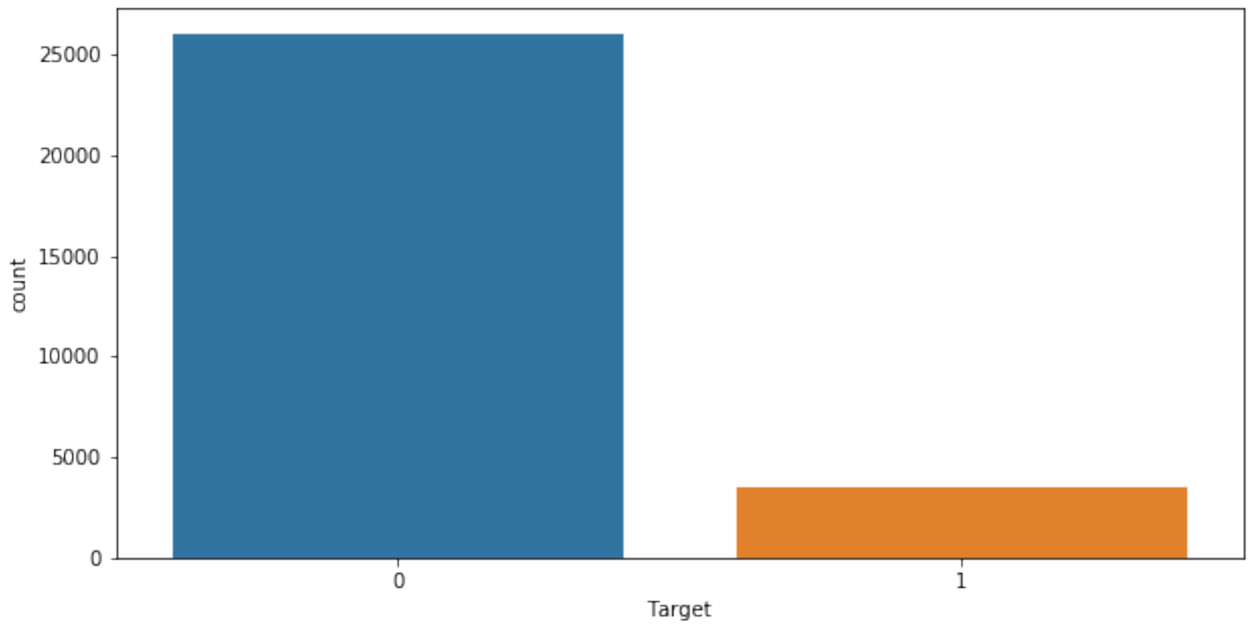
Now let's see the proportion of each target class (make over 50k a year or not).

In [7]:

```
# The proportion of each target class
NotOver50k,Over50k = train_data.Target.value_counts()
print(f'NotOver50k {NotOver50k}')
print(f'Over50k {Over50k}')
print(f'Over50k proportion {round((100*Over50k/(Over50k+NotOver50k)),2)}%')
plt.figure(figsize=(10,5))
sns.countplot(train_data['Target'])
NotOver50k 26008
Over50k 3506
Over50k proportion 11.88%
```

Out[7]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x11c7ff710>
```



According to above calculation and plot, there are only 11.88% samples makes over 50K a year.

Therefore, our training dataset is quiet imbalanced.

Now, let's do **explorative data analysis** for numerical features in our training dataset.

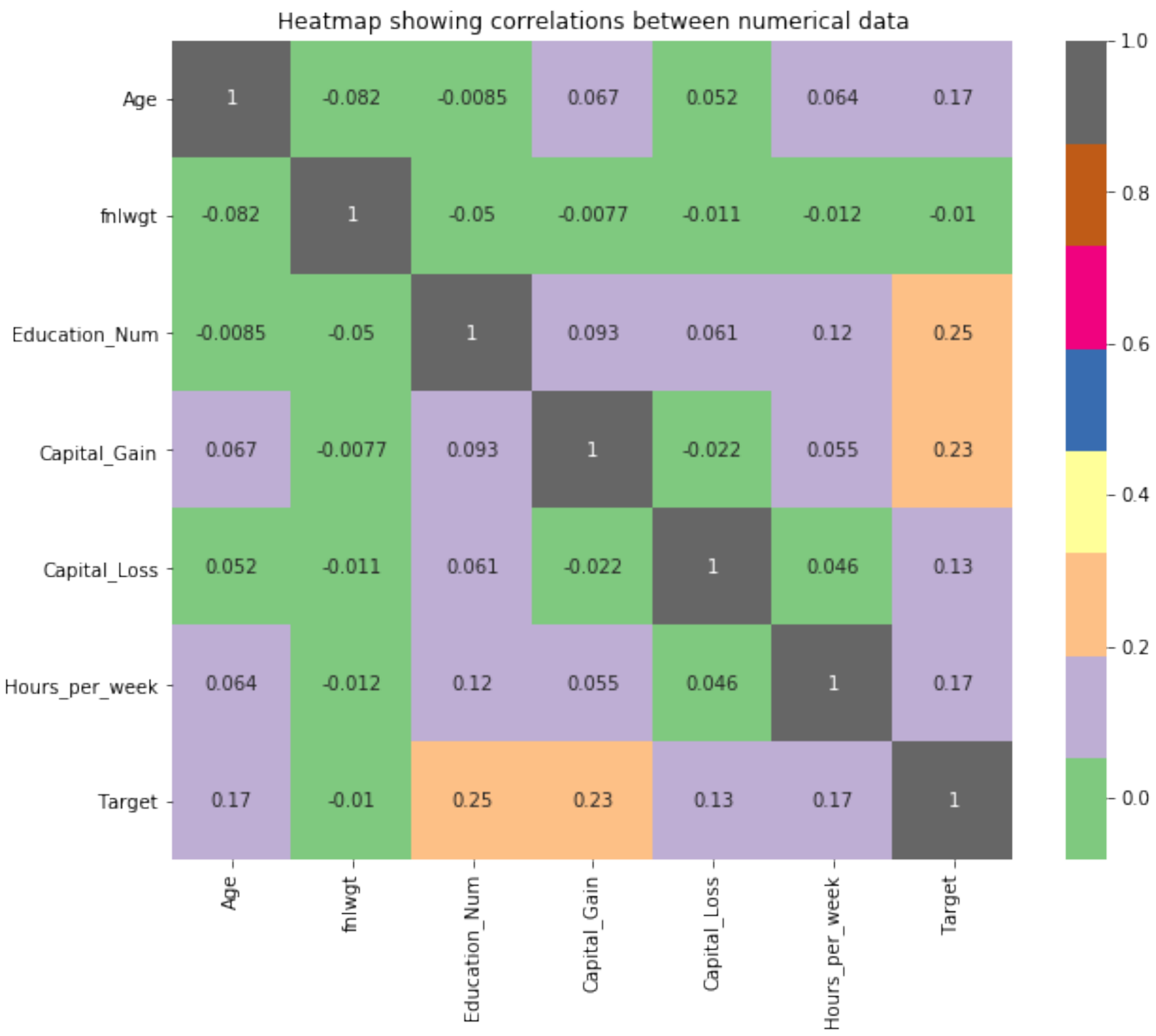
As following, we analyze the correlation coefficients between our numerical features.

In [8]:

```
# EDA for numerical features
# data.corr()
plt.figure(figsize=(10,8))
sns.heatmap(train_data.corr(),cmap='Accent',annot=True)
plt.title('Heatmap showing correlations between numerical data')
```

Out[8]:

```
Text(0.5, 1, 'Heatmap showing correlations between numerical data')
```



	Age	fnlwgt	Education_Num	Capital_Gain	Capital_Loss	Hours_per_week	Target
Age	1	-0.082	-0.0085	0.067	0.052	0.064	0.17
fnlwgt	-0.082	1	-0.05	-0.0077	-0.011	-0.012	-0.01
Education_Num	-0.0085	-0.05	1	0.093	0.061	0.12	0.25
Capital_Gain	0.067	-0.0077	0.093	1	-0.022	0.055	0.23
Capital_Loss	0.052	-0.011	0.061	-0.022	1	0.046	0.13
Hours_per_week	0.064	-0.012	0.12	0.055	0.046	1	0.17
Target	0.17	-0.01	0.25	0.23	0.13	0.17	1

One thing to note is that the correlation coefficient between 'fnlwgt' and our target is quiet small (which is -0.01).

Therefore, I don't consider 'fnlwgt' in my NN models.

(In fact, I have tried to incorporate 'fnlwgt' in my NN models and got really bad results.)

In addition, because I think 'Education' and 'Education_Num' contain the same information, I only use 'Education_Num' in my NN models.

Now, let's do the **explorative data analysis** for some categorical features.

In [9]:

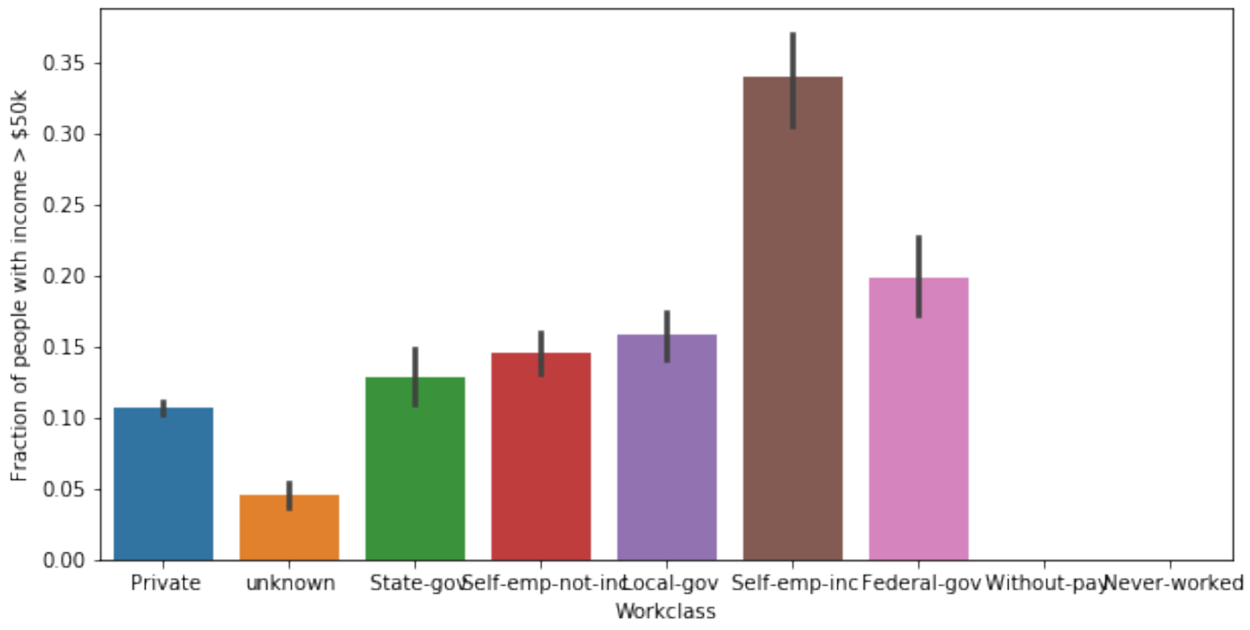
```
# Drop 'fnlwgt' & 'Education'
train_data = train_data.drop(columns=['fnlwgt','Education'])
X_test = X_test.drop(columns=['fnlwgt','Education'])
```

In [10]:

```
# EDA for categorical features
plt.figure(figsize=(10,5))
ax = sns.barplot(x='Workclass',y='Target',data=train_data)
ax.set(ylabel='Fraction of people with income > $50k')
```

Out[10]:

```
[Text(0, 0.5, 'Fraction of people with income > $50k')]
```



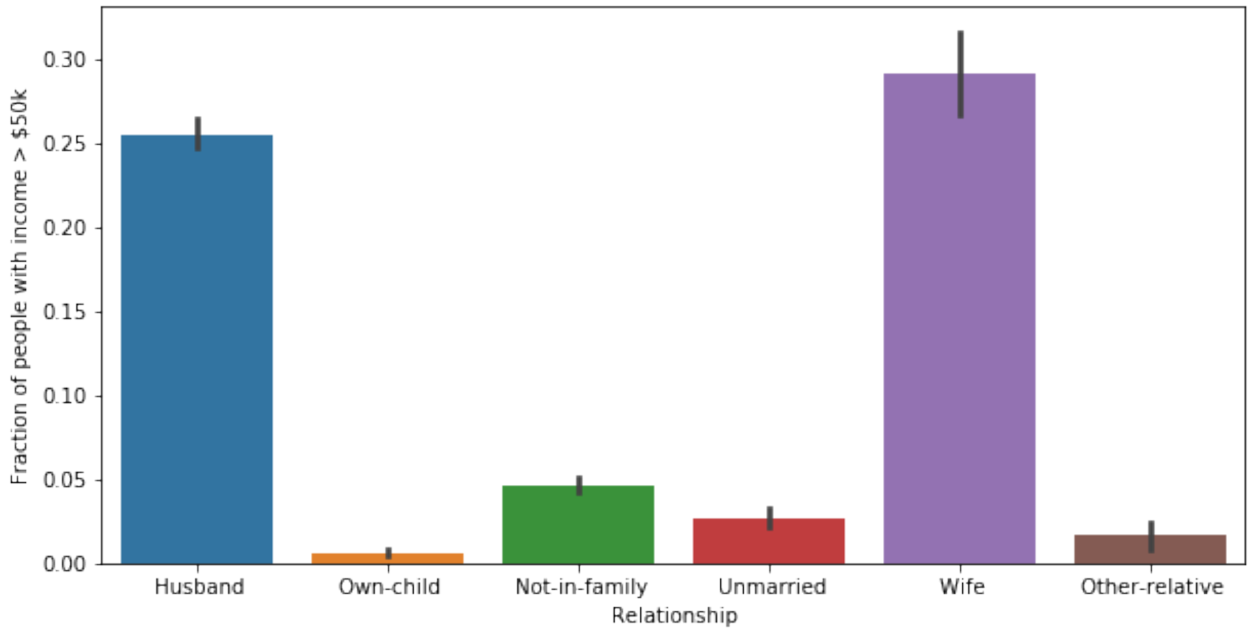
```
As above, people who are 'Self-emp-inc' are more likely makes over 50K a year.
```

In [11]:

```
plt.figure(figsize=(10,5))
ax = sns.barplot(x='Relationship',y='Target',data=train_data)
ax.set(ylabel='Fraction of people with income > $50k')
```

Out[11]:

```
[Text(0, 0.5, 'Fraction of people with income > $50k')]
```



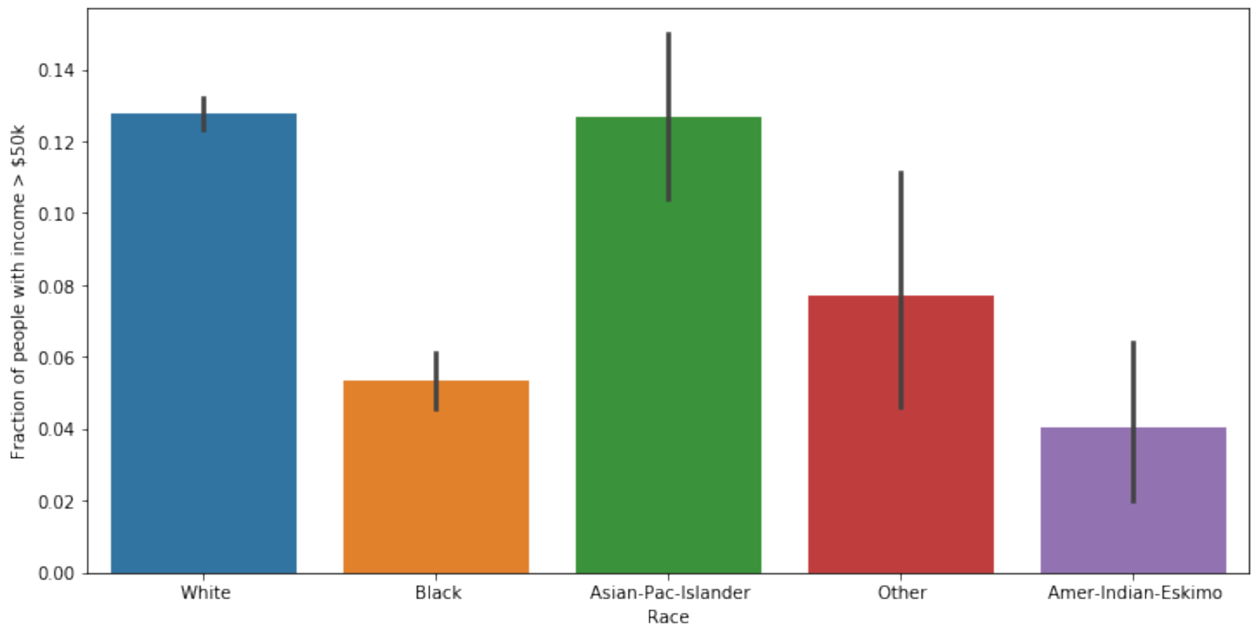
As above, 'Husband' and 'Wife' are more likely makes over 50K a year.

In [12]:

```
plt.figure(figsize=(12,6))
ax=sns.barplot(x='Race',y='Target',data=train_data)
ax.set(ylabel='Fraction of people with income > $50k')
```

Out[12]:

```
[Text(0, 0.5, 'Fraction of people with income > $50k')]
```



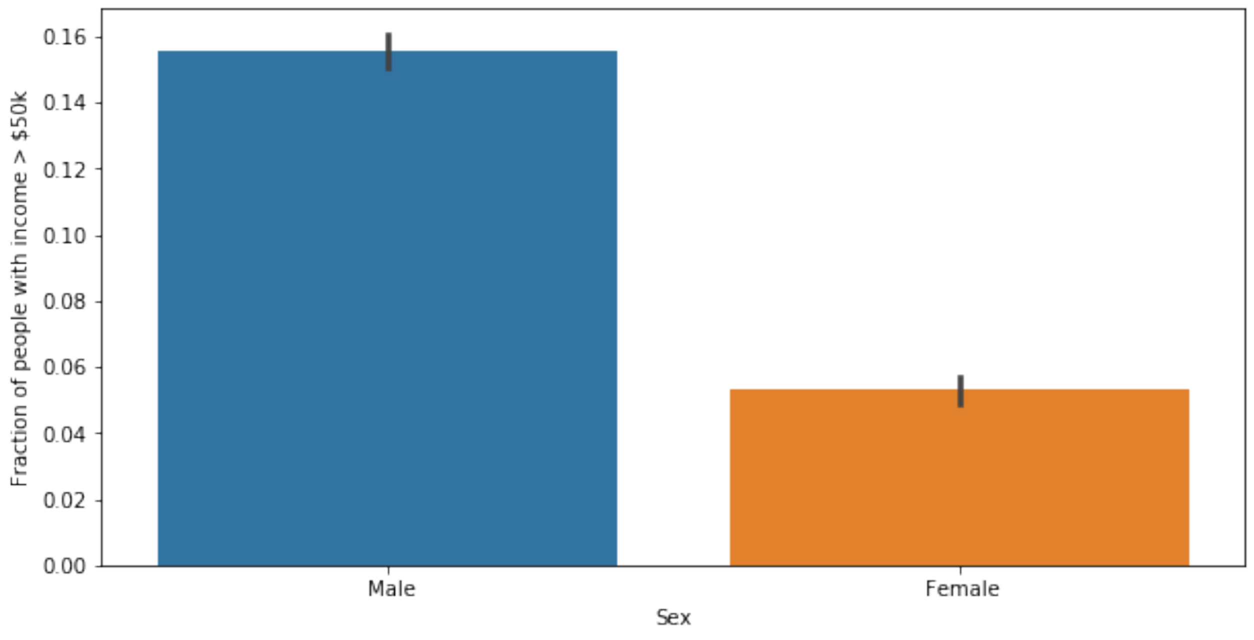
As above, 'White' and 'Asian-Pac-Islander Race' have higher proportion of people who make over 50K a year.

In [13]:

```
plt.figure(figsize=(10,5))
ax = sns.barplot(x='Sex',y='Target',data=train_data)
ax.set(ylabel='Fraction of people with income > $50k')
```

Out[13]:

```
[Text(0, 0.5, 'Fraction of people with income > $50k')]
```



As above, 'Male' group has higher proportion of people who make over 50K a year.

Now, we split our training data into features (X) and label (y).

In [14]:

```
# Split our train_data
X_train=train_data.iloc[:,:-1]
y_train=train_data.iloc[:,-1]
```

Now, we have to deal with the issue of categorical features.

In our training data and testing data, we have many categorical features ('Workclass','Martial_Status', 'Education', 'Occupation','Relationship','Race','Sex','Country'). Because our NN models can only deal with numbers, we have to encode these categorical features into numbers.

In fact, there are many different ways to **encode categorical features**. The method I used is so-called 'One-hot encoding' (as following).

In [15]:

```
# Use 'One-hot encoding' to encode categorical features.
X = X_train.append(X_test)
X = pd.get_dummies(X)
X_train = X[:29514]
X_test = X[29514:]
```

Now, we transform our datasets from dataframes to arrays, so we can feed them into NN models.

In addition, we use 7500 samples in our training dataset to be our validation set and use other samples to train our NN models.

In [16]:

```
# Change dataframes to arrays
X_train = np.asarray(X_train)
X_test = np.asarray(X_test)
y_train = np.asarray(y_train).astype('float32')

# validation set
X_valid = X_train[:7500]
partial_X_train = X_train[7500:]
y_valid = y_train[:7500]
partial_y_train = y_train[7500:]
```

So far, we have dealed with all issues of data pre-processing.

Now, we can start to build our NN models.

DL model draft:

3 hidden layers in this model.

The first hidden layer: 16 units with 'relu' activation function.

The second hidden layer: 16 units with 'relu' activation function.

The third hidden layer: 1 unit with 'sigmoid' activation function.

I choose the 'rmsprop' optimizer, 'binary_crossentropy' loss function, and the 'accuracy' metrics.

parameter initialization:

I use 200 epochs to train my model. The batch_size of my model is 512.

parameter tuning:

Because I think this model did well in my training dataset, I didn't tune it's parameters.

To begin with, we build our NN model with very simple structure as following.

In [17]:

```
# Construct our model
from keras import models
from keras import layers

model = models.Sequential()
model.add(layers.Dense(16, activation='relu'))
model.add(layers.Dense(16, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))

model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['accuracy'])
Using TensorFlow backend.
```

Now we can start to fit our NN model and record all information in 'history'.

In [18]:

```
# Iterate on your training data by calling the fit() method of your model
history = model.fit(partial_X_train,
                    partial_y_train,
                    epochs=200,
                    batch_size=512,
                    validation_data=(X_valid, y_valid))
```

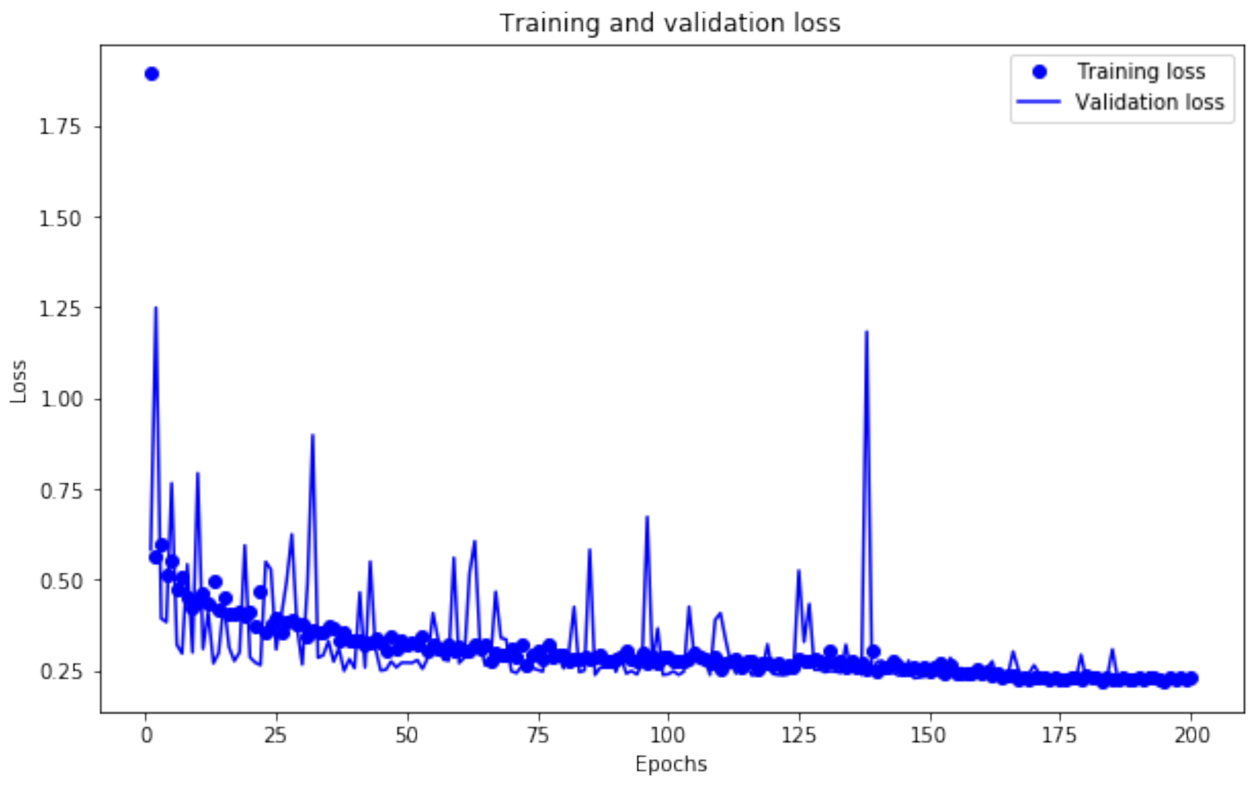
```
Train on 22014 samples, validate on 7500 samples
Epoch 1/200
22014/22014 [==============================] - 0s 16us/step - loss: 1.8925 -
accuracy: 0.8715 - val_loss: 0.5837 - val_accuracy: 0.8764
Epoch 2/200
22014/22014 [==============================] - 0s 7us/step - loss: 0.5647 -
accuracy: 0.8739 - val_loss: 1.2489 - val_accuracy: 0.8592
Epoch 3/200
22014/22014 [==============================] - 0s 10us/step - loss: 0.6010 -
accuracy: 0.8739 - val_loss: 0.3940 - val_accuracy: 0.8836
Epoch 4/200
22014/22014 [==============================] - 0s 8us/step - loss: 0.5143 -
accuracy: 0.8804 - val_loss: 0.3838 - val_accuracy: 0.8935
Epoch 5/200
22014/22014 [==============================] - 0s 11us/step - loss: 0.5509 -
accuracy: 0.8828 - val_loss: 0.7664 - val_accuracy: 0.8792
.
.
.
Epoch 195/200
22014/22014 [==============================] - 0s 6us/step - loss: 0.2210 -
accuracy: 0.9093 - val_loss: 0.2265 - val_accuracy: 0.9104
Epoch 196/200
22014/22014 [==============================] - 0s 6us/step - loss: 0.2318 -
accuracy: 0.9109 - val_loss: 0.2276 - val_accuracy: 0.9113
Epoch 197/200
22014/22014 [==============================] - 0s 6us/step - loss: 0.2249 -
accuracy: 0.9102 - val_loss: 0.2275 - val_accuracy: 0.9112
Epoch 198/200
22014/22014 [==============================] - 0s 6us/step - loss: 0.2309 -
accuracy: 0.9082 - val_loss: 0.2325 - val_accuracy: 0.9100
Epoch 199/200
22014/22014 [==============================] - 0s 6us/step - loss: 0.2237 -
accuracy: 0.9099 - val_loss: 0.2280 - val_accuracy: 0.9119
Epoch 200/200
22014/22014 [==============================] - 0s 6us/step - loss: 0.2298 -
accuracy: 0.9081 - val_loss: 0.2292 - val_accuracy: 0.9093
```

Now, we can plot the results of loss values from the training and validation set.

In [19]:

```
# plot the results of loss values from the training set and validtion set
history_dict = history.history
loss_values = history_dict['loss']
val_loss_values = history_dict['val_loss']

epochs = range(1, len(history_dict['accuracy']) + 1)
```

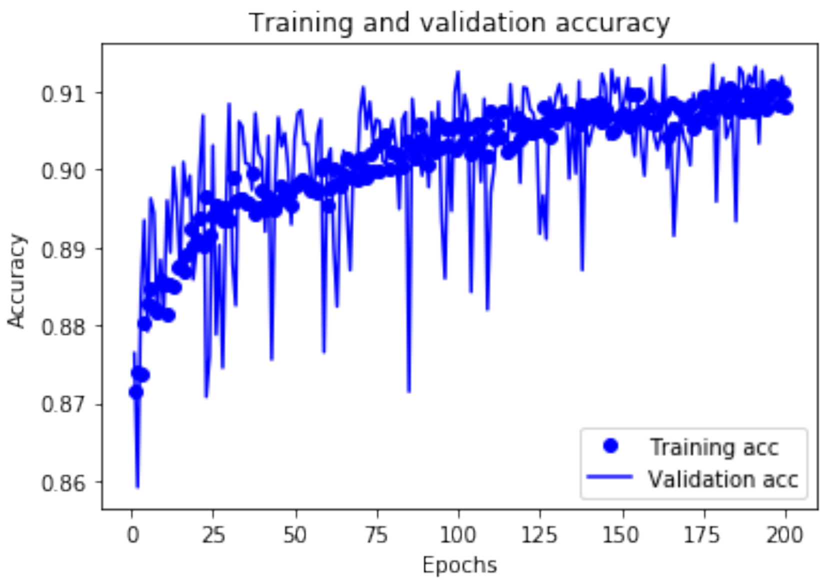
```
plt.figure(figsize=(10,6))
plt.plot(epochs, loss_values, 'bo', label='Training loss')
plt.plot(epochs, val_loss_values, 'b', label='Validation loss')
plt.title('Training and validation loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()
```



Now, we can plot the results of accuracy from the training and validation set.

In [20]:

```
# plot the results of accuracy from the training set and validtion set
acc = history_dict['accuracy']
val_acc = history_dict['val_accuracy']
plt.plot(epochs, acc, 'bo', label='Training acc')
plt.plot(epochs, val_acc, 'b', label='Validation acc')
plt.title('Training and validation accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
plt.show()
```



This simple works well in our training set and validation set(with accuracy about 0.9).

However, this model didn't perform well on our test set. When I used all the training data to train this model, the public score of this model is about 0.62. Why this happened? Well, I think the reason is about our imbalanced data, there are too many 0s in 'Target'. As a result, our model prefer to give us many 0s, which leads to our fail on testing dataset.

Now, in order to deal with the issue of imbalanced dataset, I use a weighted model.

To begin with, let's calculate class_weights.

In [21]:

```
# Calculate class weight
NotOver50k, Over50k = np.bincount(train_data.Target)
total_count = len(train_data.Target)

weight_no_over50k = (1/NotOver50k)*(total_count)/2.0
weight_over50k = (1/Over50k)*(total_count)/2.0

class_weights = {0:weight_no_over50k, 1:weight_over50k}
```

Now, we can use class_weights as an argument when we construct our second NN model.

DL model draft:

3 hidden layers in this model.

The first hidden layer: 16 units with 'relu' activation function.

The second hidden layer: 16 units with 'relu' activation function.

The third hidden layer: 1 unit with 'sigmoid' activation function.

I choose the 'rmsprop' optimizer, 'binary_crossentropy' loss function, and the 'accuracy' metrics.

parameter initialization:

I use 200 epochs to train my model. The batch_size of my model is 512.

parameter tuning:

Because I think this model did well in my training dataset, I didn't tune it's parameters.

Now, let's construct our second NN model.

In [22]:

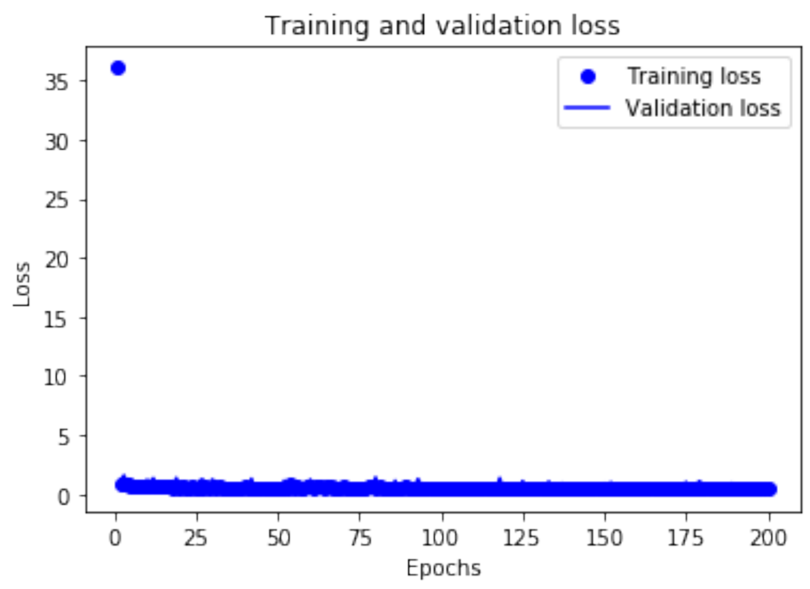
```
# Use weighted model!
model = models.Sequential()
model.add(layers.Dense(16, activation='relu'))
model.add(layers.Dense(16, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))

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

history = model.fit(partial_X_train,
                    partial_y_train,
                    epochs=200,
                    batch_size=512,
                    validation_data=(X_valid, y_valid),
                    class_weight=class_weights)
Train on 22014 samples, validate on 7500 samples
Epoch 1/200
22014/22014 [==============================] - 0s 19us/step - loss: 36.1078 -
accuracy: 0.6971 - val_loss: 0.7639 - val_accuracy: 0.7001
Epoch 2/200
22014/22014 [==============================] - 0s 12us/step - loss: 0.7924 -
accuracy: 0.7053 - val_loss: 1.1875 - val_accuracy: 0.7517
Epoch 3/200
22014/22014 [==============================] - 0s 11us/step - loss: 0.7911 -
accuracy: 0.7337 - val_loss: 1.5703 - val_accuracy: 0.7423
Epoch 4/200
22014/22014 [==============================] - 0s 9us/step - loss: 0.8578 -
accuracy: 0.7393 - val_loss: 0.8123 - val_accuracy: 0.7447
Epoch 5/200
```

```
22014/22014 [==============================] - 0s 15us/step - loss: 0.6984 -
accuracy: 0.7435 - val_loss: 1.2160 - val_accuracy: 0.7577
.
.
.
Epoch 195/200
22014/22014 [==============================] - 0s 6us/step - loss: 0.4190 -
accuracy: 0.7978 - val_loss: 0.4612 - val_accuracy: 0.7912
Epoch 196/200
22014/22014 [==============================] - 0s 6us/step - loss: 0.4294 -
accuracy: 0.8030 - val_loss: 0.4430 - val_accuracy: 0.7957
Epoch 197/200
22014/22014 [==============================] - 0s 6us/step - loss: 0.4873 -
accuracy: 0.7936 - val_loss: 0.3824 - val_accuracy: 0.8125
Epoch 198/200
22014/22014 [==============================] - 0s 6us/step - loss: 0.4047 -
accuracy: 0.8053 - val_loss: 0.3887 - val_accuracy: 0.8103
Epoch 199/200
22014/22014 [==============================] - 0s 6us/step - loss: 0.4485 -
accuracy: 0.7984 - val_loss: 0.3704 - val_accuracy: 0.8160
Epoch 200/200
22014/22014 [==============================] - 0s 6us/step - loss: 0.4922 -
accuracy: 0.7963 - val_loss: 0.4279 - val_accuracy: 0.8055
```

After constructing our weighted model, we can plot the results of loss values from the training and validation set.

In [23]:

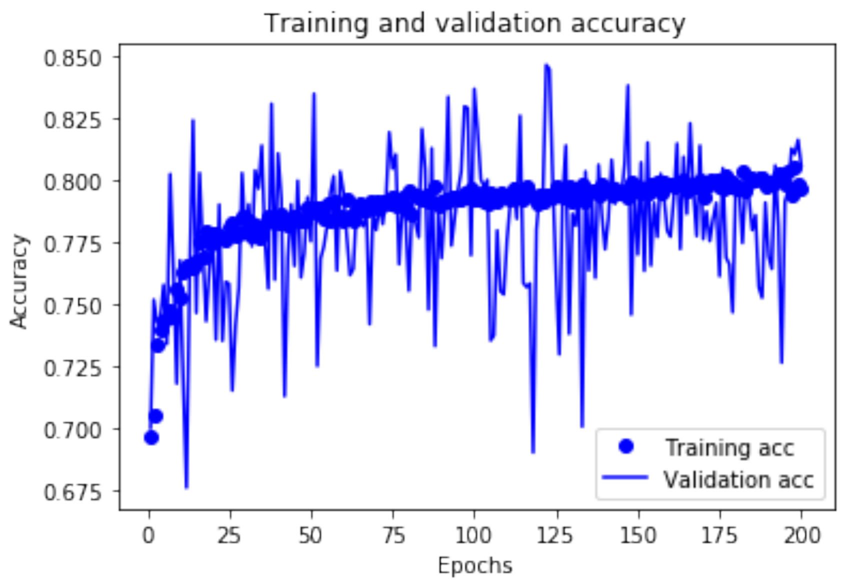
```
# plot the results of loss values from the training set and validtion set
history_dict = history.history
loss_values = history_dict['loss']
val_loss_values = history_dict['val_loss']

epochs = range(1, len(history_dict['accuracy']) + 1)

plt.plot(epochs, loss_values, 'bo', label='Training loss')
plt.plot(epochs, val_loss_values, 'b', label='Validation loss')
plt.title('Training and validation loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()
```



Also, we can plot the results of accuracy from the training and validation set.

In [24]:

```
#plt.clf()
acc = history_dict['accuracy']
val_acc = history_dict['val_accuracy']
plt.plot(epochs, acc, 'bo', label='Training acc')
plt.plot(epochs, val_acc, 'b', label='Validation acc')
plt.title('Training and validation accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
plt.show()
```



When I used all the training data to train this model, the public score of this model is about 0.83 (the private score is also about 0.82). By using weighted model, we successfully overcome the problem of imbalanced training dataset.

Now, I can use all the training data to train this model.

In [25]:

```
# Training the final model
model = models.Sequential()
model.add(layers.Dense(16, activation='relu'))
model.add(layers.Dense(16, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))

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

model.fit(X_train,y_train,
                    epochs=200,
                    batch_size=512,
                    class_weight=class_weights)
Epoch 1/200
29514/29514 [==============================] - 0s 11us/step - loss: 2.4969 -
accuracy: 0.2443
Epoch 2/200
29514/29514 [==============================] - 0s 5us/step - loss: 1.0225 -
accuracy: 0.7211
Epoch 3/200
29514/29514 [==============================] - 0s 6us/step - loss: 0.7428 -
accuracy: 0.7674
Epoch 4/200
29514/29514 [==============================] - 0s 5us/step - loss: 0.7877 -
accuracy: 0.7632
Epoch 5/200
29514/29514 [==============================] - 0s 5us/step - loss: 0.8176 -
accuracy: 0.7551
.
.
.
Epoch 195/200
29514/29514 [==============================] - 0s 6us/step - loss: 0.3987 -
accuracy: 0.7987
Epoch 196/200
29514/29514 [==============================] - 0s 6us/step - loss: 0.4506 -
accuracy: 0.7935
Epoch 197/200
29514/29514 [==============================] - 0s 6us/step - loss: 0.4081 -
accuracy: 0.7981
Epoch 198/200
29514/29514 [==============================] - 0s 6us/step - loss: 0.4105 -
accuracy: 0.7986
Epoch 199/200
```

```
29514/29514 [==============================] - 0s 6us/step - loss: 0.4280 -
accuracy: 0.7954
Epoch 200/200
29514/29514 [==============================] - 0s 6us/step - loss: 0.4013 -
accuracy: 0.7990
```

Out[25]:

```
<keras.callbacks.callbacks.History at 0x11cbbb090>
```

After training our final model, we can then use this model to predict our final answer (use our test dataset).

In [26]:

```
#Using a trained network to generate predictions on new data
y_pred_probability=model.predict(X_test)
y_pred=(y_pred_probability>0.5).astype(int)
answer=pd.DataFrame(y_pred)
```

After some dataframe operations, we can then export our answer.

Now, let's see the prediction result of our final model.

The prediction result:

As we talked before, by using our final weighted model, in Kaggle competition, the public score of this model is about 0.83 (the private score is also about 0.82).

In my opinion, this model's great performance in Kaggle competition means this model can effectively predict whether a person makes over 50K a year. That is to say, if we want to predict if a person makes over 50K a year, our final model is trustable and reasonable.

Learning progress and reflection:

To be honest, when I first built a NN model for this assignment, the result was very bad. Even if I used the weighted model technique, I still got bad results (accuracy scores are very unstable).

In order to overcome this situation, I started to do explorative data analysis and I found that 'fnlwgt' is almost unrelated with 'Target'.

Therefore, I decided to drop 'fnlwgt'. Fortunately, my models started to improve and got trustable predictions.

In short, never forget to do EDA before modeling.

Feedback for the teaching team's reference:

Thanks for your teaching and helping. I have learned a lot from this journey.