Homework 1 by Yan Liu

In [1]:
```
# Import libraries

import nltk
#nltk.download('udhr')
#nltk.download('inaugural')
from nltk.corpus import udhr
from nltk.corpus import inaugural
from nltk.corpus import wordnet as wn
import matplotlib as pltlib
pltlib.rcParams['figure.dpi'] = 900
import matplotlib.pyplot as plt
plt.style.use('ggplot')

import numpy as np
import pandas as pd
import itertools
%matplotlib inline

#this library is used for splitting to training and validation set
from sklearn.model_selection import train_test_split

#to encode the class label in one hot encoded style
from sklearn.preprocessing import OneHotEncoder

#model architecture library
from keras.models import Sequential
from keras import layers

#for the optimization
from keras.optimizers import RMSprop,Adam,SGD
#for adding layers
from keras.layers import Dense, Activation, Dropout
#from keras.callbacks import ModelCheckpoint

#for encoding in one hot encoded style
from keras.utils import to_categorical

import random
from sklearn.metrics import confusion_matrix
```

```
Using TensorFlow backend.
```

Problem 1

Use the text of the Universal Declaration of Human Rights (UDHR). Create a table for 4 languages of your choice. Use that table to collect statistics about those languages. Place in that table the number of words in UDHR in each language, number of unique words, average length of words, number of sentences contained in UDHR and average number of words per sentence. You do not have to populate the table from your code. You may, but you may also determine individual values separately and enter them in the table manually. Create a distribution of sentence lengths for all four language. Distribution of sentence lengths presents the number of sentences of varying length. Plot those (non-cumulative) distributions for all four languages using one diagram. (25%)

In [2]:
```
# Select four languages
languages = ['French_Francais', 'English', 'German_Deutsch','Spanish']
num_words = [len(udhr.words(lang + '-Latin1')) for lang in languages]
num_unique_words = [len(set(udhr.words(lang + '-Latin1'))) for lang in languages]
num_sents = [len(udhr.sents(lang + '-Latin1')) for lang in languages]
```

In [3]:
```
avg_word_len = [round(sum([len(w) for w in udhr.words("English" + '-Latin1')])/num_word,3) for lang,num_word in zip(languages, num_words)]
```

In [4]:
```
avg_unique_word_len = [round(sum([len(w) for w in set(udhr.words("English" + '-Latin1'))])/num_word,3) for lang,num_word in zip(languages, num_unique_words)]
```

In [5]:
```
avg_word_per_sent = []
for lang in languages:
    avg_word_per_sent.append(len(sent) for sent in udhr.sents(lang + '-Latin1'))
```

In [6]:
```
words_cnt_per_sent = {}
avg_word_per_sent = []

for lang in languages:
    sents = udhr.sents(lang + '-Latin1')
    words_cnt_per_sent[lang] = [len(sent) for sent in sents]
    avg_word_per_sent.append(round(sum(words_cnt_per_sent[lang])/len(sents),3))
```

In [7]:
```
# Create a data frame to showcase the results
table = pd.DataFrame({
                    "Lanugages" : languages,
                    "Number of Words" :num_words,
                    "Number of Unique Words" : num_unique_words,
                    "Average Length of Words" : avg_word_len,
                    "Number of Sentences" :  num_sents,
                    "Average Number of Words per Sentence" : avg_w
ord_per_sent
                    })
```

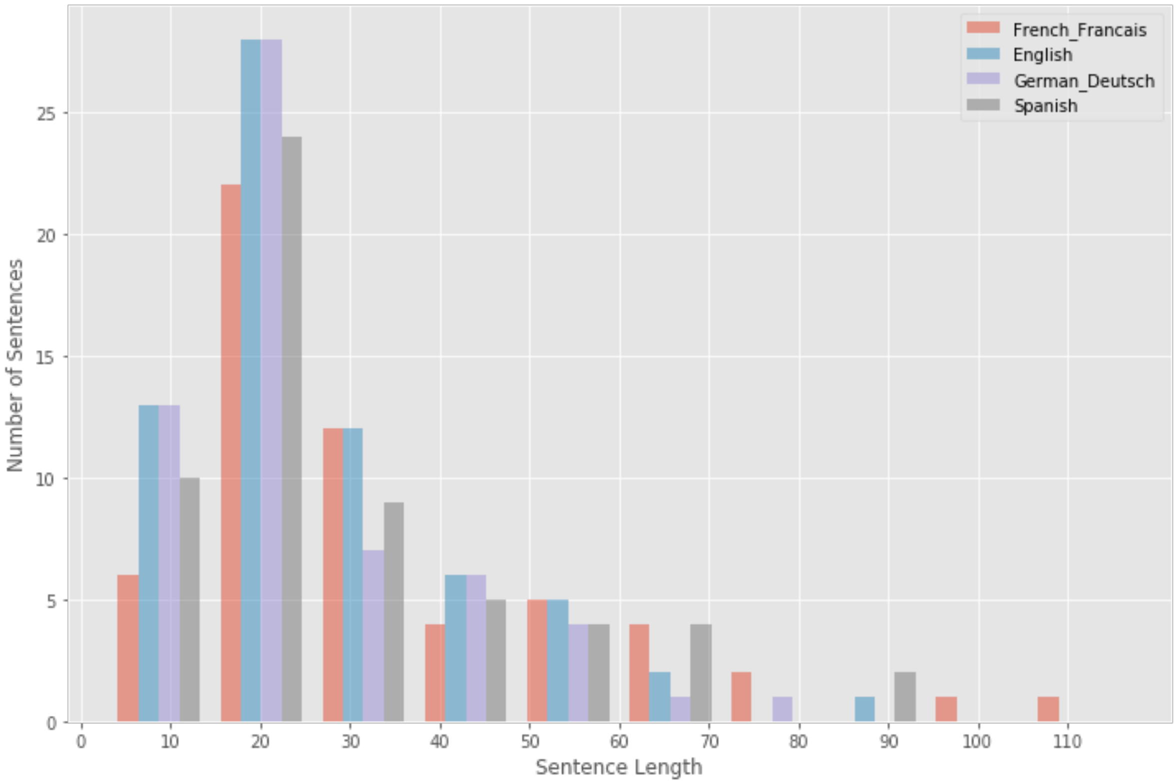
In [8]:
```
table
```

Out[8]:

	Lanugages	Number of Words	Number of Unique Words	Average Length of Words	Number of Sentences	Average Number of Words per Sentence
0	French_Francais	1935	567	4.274	57	33.947
1	English	1781	533	4.644	67	26.582
2	German_Deutsch	1521	579	5.438	60	25.350
3	Spanish	1763	542	4.691	58	30.397

Create a distribution of sentence lengths for all four language. Distribution of sentence lengths presents the number of sentences of varying length. Plot those (non-cumulative) distributions for all four languages using one diagram.

In [9]:
```
fig, ax = plt.subplots(figsize=(12,8))
plt.hist(list(words_cnt_per_sent.values()), alpha = .5, bins = 10,
label = languages)
plt.xlabel('Sentence Length')
plt.ylabel('Number of Sentences')
plt.legend(loc='upper right')
ax.set_xticks(range(0,120,10))
plt.show()
```



In general, the distribution of sentence length is approximately the same for the selected four lanugages. All the four languages have most sentences of length 10 to 20. French tends to be verbose as it has 1 or more sentences whose length fall into the [100,110] interval.

Problem 2.

Identify 10 most frequently used words longer than 7 characters in the entire corpus of Inaugural Addresses. Do not identify 10 words for every speech but rather 10 words for the entire corpus. Which among those words has the largest number of synonyms? List all synonyms for those 10 words. Which one of those 10 words has the largest number of hyponyms? List all hyponyms of those 10 most frequently used “long” words. The purpose of this problem is to familiarize you with WordNet and concepts of synonyms and hyponyms. (25%)

In [10]:
```
# Load and concat all inaugurals words
inaugurals = [inaugural.words(fileid) for fileid in inaugural.filei
ds()]
concat_words = list(itertools.chain.from_iterable(inaugurals))
```

10 most frequently used words longer than 7 characters

In [11]:
```
wordsDist = nltk.FreqDist(w.lower() for w in concat_words if len(w)
> 7)
```

In [12]:
```
wordsDist.most_common(10)
```

Out[12]:
```
[('government', 600),
 ('citizens', 247),
 ('constitution', 206),
 ('american', 163),
 ('national', 157),
 ('congress', 130),
 ('interests', 115),
 ('political', 106),
 ('executive', 97),
 ('principles', 96)]
```

In [13]:
```
# Create a dictionary with the 10 words as keys and their synonym l
ist as values
words10 = [t[0] for t in wordsDist.most_common(10)]
word_synonym = {}
for w in words10:
    synonyms = []
    for synset in wn.synsets(w):
        for lemma_name in synset.lemma_names():
            if lemma_name not in synonyms:
                synonyms.append(lemma_name)
    word_synonym[w] = synonyms
```

List all synonyms for those 10 words

In [14]:
```
word_synonym
```

Out[14]:
```
{'government': ['government',
  'authorities',
  'regime',
  'governing',
  'governance',
  'government_activity',
  'administration',
  'politics',
  'political_science'],
 'citizens': ['citizen'],
 'constitution': ['fundamental_law',
  'organic_law',
  'constitution',
  'establishment',
  'formation',
  'organization',
  'organisation',
  'United_States_Constitution',
```

```
  'U.S._Constitution',
  'US_Constitution',
  'Constitution',
  'Constitution_of_the_United_States',
  'composition',
  'physical_composition',
  'makeup',
  'make-up',
  'Old_Ironsides'],
 'american': ['American', 'American_English', 'American_language']
,
 'national': ['national', 'subject', 'home', 'interior', 'internal
'],
 'congress': ['Congress',
  'United_States_Congress',
  'U.S._Congress',
  'US_Congress',
  'congress',
  'sexual_intercourse',
  'intercourse',
  'sex_act',
  'copulation',
  'coitus',
  'coition',
  'sexual_congress',
  'sexual_relation',
  'relation',
  'carnal_knowledge'],
 'interests': ['interest',
  'involvement',
  'sake',
  'interestingness',
  'stake',
  'interest_group',
  'pastime',
  'pursuit',
  'concern',
  'occupy',
  'worry',
  'matter_to'],
 'political': ['political'],
 'executive': ['executive', 'executive_director', 'administrator']
,
 'principles': ['principle', 'rule', 'precept', 'rationale']}
```

Words has the largest number of synonyms - constitution

In [15]:
```
sorted([(k,len(v)) for k,v in word_synonym.items()], key=lambda x: x[1], reverse = True)
```

Out[15]:
```
[('constitution', 17),
 ('congress', 15),
 ('interests', 12),
 ('government', 9),
 ('national', 5),
 ('principles', 4),
 ('american', 3),
 ('executive', 3),
 ('citizens', 1),
 ('political', 1)]
```

In [16]:
```
word_hyponym = {}
for w in words10:
    hyponyms = []
    for synset in wn.synsets(w):
        for lemma_name in synset.hyponyms():
            if lemma_name not in hyponyms:
                hyponyms.append(lemma_name)
    word_hyponym[w] = hyponyms
```

List all hyponyms of those 10 most frequently used “long” words.

In [17]:
```
word_hyponym
```

Out[17]:
```
{'government': [Synset('ancien_regime.n.01'),
  Synset('authoritarian_state.n.01'),
  Synset('bureaucracy.n.02'),
  Synset('court.n.03'),
  Synset('downing_street.n.02'),
  Synset('empire.n.02'),
  Synset('federal_government.n.01'),
  Synset('government-in-exile.n.01'),
  Synset('local_government.n.01'),
  Synset('military_government.n.01'),
  Synset('palace.n.02'),
  Synset('papacy.n.01'),
  Synset('puppet_government.n.01'),
  Synset('state.n.03'),
  Synset('state_government.n.01'),
  Synset('totalitarian_state.n.01'),
  Synset('legislation.n.02'),
  Synset('misgovernment.n.01'),
  Synset('trust_busting.n.01'),
  Synset('geopolitics.n.01'),
  Synset('realpolitik.n.01')],
 'citizens': [Synset('active_citizen.n.01'),
  Synset('civilian.n.01'),
  Synset('freeman.n.01'),
  Synset('private_citizen.n.01'),
  Synset('repatriate.n.01'),
  Synset('thane.n.02'),
```

```
  Synset('voter.n.01')],
 'constitution': [Synset('collectivization.n.01'),
  Synset('colonization.n.01'),
  Synset('communization.n.02'),
  Synset('federation.n.03'),
  Synset('unionization.n.01'),
  Synset('genotype.n.02'),
  Synset('karyotype.n.01'),
  Synset('phenotype.n.01'),
  Synset('structure.n.02'),
  Synset('texture.n.05')],
 'american': [Synset('african-american.n.01'),
  Synset('alabaman.n.01'),
  Synset('alaskan.n.01'),
  Synset('anglo-american.n.01'),
  Synset('appalachian.n.01'),
  Synset('arizonan.n.01'),
  Synset('arkansan.n.01'),
  Synset('asian_american.n.01'),
  Synset('bay_stater.n.01'),
  Synset('bostonian.n.01'),
  Synset('californian.n.01'),
  Synset('carolinian.n.01'),
  Synset('coloradan.n.01'),
  Synset('connecticuter.n.01'),
  Synset('creole.n.02'),
  Synset('delawarean.n.01'),
  Synset('floridian.n.01'),
  Synset('franco-american.n.01'),
  Synset('georgian.n.01'),
  Synset('german_american.n.01'),
  Synset('hawaiian.n.02'),
  Synset('idahoan.n.01'),
  Synset('illinoisan.n.01'),
  Synset('indianan.n.01'),
  Synset('iowan.n.01'),
  Synset('kansan.n.01'),
  Synset('kentuckian.n.01'),
  Synset('louisianan.n.01'),
  Synset('mainer.n.01'),
  Synset('marylander.n.01'),
  Synset('michigander.n.01'),
  Synset('minnesotan.n.01'),
  Synset('mississippian.n.02'),
  Synset('missourian.n.01'),
  Synset('montanan.n.01'),
  Synset('nebraskan.n.01'),
  Synset('nevadan.n.01'),
  Synset('new_englander.n.01'),
  Synset('new_hampshirite.n.01'),
  Synset('new_jerseyan.n.01'),
  Synset('new_mexican.n.01'),
  Synset('new_yorker.n.01'),
  Synset('nisei.n.01'),
  Synset('north_carolinian.n.01'),
  Synset('north_dakotan.n.01'),
  Synset('ohioan.n.01'),
```

```
 Synset('oklahoman.n.01'),
 Synset('oregonian.n.01'),
 Synset('pennsylvanian.n.02'),
 Synset('puerto_rican.n.01'),
 Synset('rhode_islander.n.01'),
 Synset('south_carolinian.n.01'),
 Synset('south_dakotan.n.01'),
 Synset('southerner.n.01'),
 Synset('spanish_american.n.01'),
 Synset('tennessean.n.01'),
 Synset('texan.n.01'),
 Synset('tory.n.01'),
 Synset('utahan.n.01'),
 Synset('vermonter.n.01'),
 Synset('virginian.n.01'),
 Synset('washingtonian.n.01'),
 Synset('washingtonian.n.02'),
 Synset('west_virginian.n.01'),
 Synset('wisconsinite.n.01'),
 Synset('wyomingite.n.01'),
 Synset('yankee.n.01'),
 Synset('yankee.n.03'),
 Synset('african_american_vernacular_english.n.01'),
 Synset('creole.n.01'),
 Synset('latin_american.n.01'),
 Synset('mesoamerican.n.01'),
 Synset('north_american.n.01'),
 Synset('south_american.n.01'),
 Synset('west_indian.n.01')],
'national': [Synset('citizen.n.01'),
 Synset('compatriot.n.01'),
 Synset('patriot.n.01')],
'congress': [Synset('continental_congress.n.01'),
 Synset('defloration.n.02'),
 Synset('fuck.n.01'),
 Synset('hank_panky.n.01'),
 Synset('penetration.n.06'),
 Synset('unlawful_carnal_knowledge.n.01')],
'interests': [Synset('concern.n.01'),
 Synset('enthusiasm.n.03'),
 Synset('behalf.n.02'),
 Synset('charisma.n.01'),
 Synset('color.n.02'),
 Synset('newsworthiness.n.01'),
 Synset('shrillness.n.01'),
 Synset('topicality.n.01'),
 Synset('compound_interest.n.01'),
 Synset('simple_interest.n.01'),
 Synset('controlling_interest.n.01'),
 Synset('equity.n.02'),
 Synset('fee.n.02'),
 Synset('grubstake.n.01'),
 Synset('insurable_interest.n.01'),
 Synset('reversion.n.01'),
 Synset('right.n.08'),
 Synset('security_interest.n.01'),
 Synset('terminable_interest.n.01'),
```

```
 Synset('undivided_interest.n.01'),
 Synset('vested_interest.n.01'),
 Synset('special_interest.n.01'),
 Synset('vested_interest.n.02'),
 Synset('avocation.n.01'),
 Synset('absorb.v.09'),
 Synset('fascinate.v.02'),
 Synset('intrigue.v.01')],
'political': [],
'executive': [Synset('corporate_executive.n.01'),
 Synset('minister.n.02'),
 Synset('rainmaker.n.01'),
 Synset('surgeon_general.n.01'),
 Synset('vice_president.n.01'),
 Synset('bush_administration.n.01'),
 Synset('bush_administration.n.02'),
 Synset('carter_administration.n.01'),
 Synset('clinton_administration.n.01'),
 Synset('reagan_administration.n.01'),
 Synset('commissioner.n.01'),
 Synset('director_of_central_intelligence.n.01'),
 Synset('prefect.n.01'),
 Synset('secretary_general.n.01'),
 Synset('triumvir.n.01')],
'principles': [Synset('feng_shui.n.01'),
 Synset('pillar.n.01'),
 Synset('yang.n.01'),
 Synset('yin.n.01'),
 Synset('accounting_principle.n.01'),
 Synset('chivalry.n.02'),
 Synset('ethic.n.01'),
 Synset('hellenism.n.01'),
 Synset('legal_principle.n.01'),
 Synset('scruple.n.03'),
 Synset('conservation.n.03'),
 Synset('dictate.n.02'),
 Synset('fundamentals.n.01'),
 Synset('insurrectionism.n.01'),
 Synset('logic.n.03'),
 Synset('pleasure_principle.n.01'),
 Synset('reality_principle.n.01'),
 Synset('tao.n.02'),
 Synset('gestalt_law_of_organization.n.01'),
 Synset('gresham's_law.n.01'),
 Synset('le_chatelier's_principle.n.01'),
 Synset('localization_of_function.n.01'),
 Synset('mass-action_principle.n.01'),
 Synset('mass-energy_equivalence.n.01'),
 Synset('naegele's_rule.n.01'),
 Synset('occam's_razor.n.01'),
 Synset('principle_of_equivalence.n.01'),
 Synset('principle_of_liquid_displacement.n.01'),
 Synset('principle_of_superposition.n.01'),
 Synset('principle_of_superposition.n.02'),
 Synset('caveat_emptor.n.01'),
 Synset('higher_law.n.01'),
 Synset('hypothetical_imperative.n.01'),
```

```
           Synset('moral_principle.n.02'),
           Synset('dialectics.n.01')]}
```

The word that has the largest number of hyponyms - american

In [18]:
```
sorted([(k,len(v)) for k,v in word_hyponym.items()], key=lambda x:
x[1], reverse = True)
```

Out[18]:
```
[('american', 75),
 ('principles', 35),
 ('interests', 27),
 ('government', 21),
 ('executive', 15),
 ('constitution', 10),
 ('citizens', 7),
 ('congress', 6),
 ('national', 3),
 ('political', 0)]
```

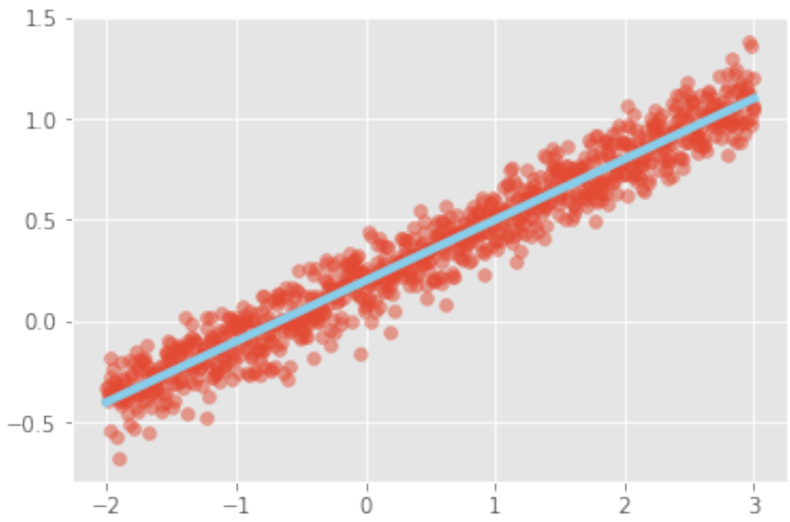
Problem 3.

Consider 100 points along the straight line in (x,y) plane represented by the linear equation y = 0.3x + 0.2. Distribute those points along the line uniformly in the interval between -2.0 and 3.0. To the y coordinate of each point add a random normally distributed value with standard deviation of 1 and mean 0. You have created and artificial set of random measurements. Create a shallow neural network with one layer which will be able to predict y value corresponding to any x value in the above interval. Implement and train the network using Keras API. Report on the accuracy of your model. This is a rather trivial problem and you do not need neural networks to solve it. We are practicing Keras API.

In [19]:
```
random.seed(100)
# Generate data points, use 1000 data points instead of 100 for hig
her accuracy
num_points=1000
x = np.linspace(-2,3,num_points)
# Add noise, scale is changed to 0.1 to lower noise
noise = np.random.normal(loc=0.0, scale=0.1, size=num_points)
y = 0.3*x + 0.2 + noise
```

In [20]:
```
# Take a look
plt.scatter(x,y,alpha =.5)
plt.plot(x,0.3*x + 0.2, color = "skyblue", linewidth = 4)
```

Out[20]: [<matplotlib.lines.Line2D at 0x145d81b38>]



In [21]:
```
# MODEL IS BUILD HERE
# 1 LAYER SIMPLE NEURAL NETWORK
#############################################

#we add the layers in sequential form
one_model = Sequential()

#first hidden layer with 20 neurons
one_model.add(Dense(20, input_shape=(1,)))

#relu nonlinear activation function is chosen
one_model.add(Activation('relu'))

#output layer
one_model.add(Dense(1))

#Use a linear activation function is chosen
one_model.add(Activation('linear'))

#show the whole model architecture
one_model.summary()
############################################
```

```
WARNING: Logging before flag parsing goes to stderr.
W0702 23:45:31.640747 4672939456 deprecation_wrapper.py:119] From
/anaconda3/envs/pytf/lib/python3.6/site-packages/keras/backend/ten
sorflow_backend.py:74: The name tf.get_default_graph is deprecated
. Please use tf.compat.v1.get_default_graph instead.

W0702 23:45:31.653775 4672939456 deprecation_wrapper.py:119] From
/anaconda3/envs/pytf/lib/python3.6/site-packages/keras/backend/ten
sorflow_backend.py:517: The name tf.placeholder is deprecated. Ple
ase use tf.compat.v1.placeholder instead.

W0702 23:45:31.656318 4672939456 deprecation_wrapper.py:119] From
/anaconda3/envs/pytf/lib/python3.6/site-packages/keras/backend/ten
sorflow_backend.py:4138: The name tf.random_uniform is deprecated.
Please use tf.random.uniform instead.


_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 20)                40
_________________________________________________________________
activation_1 (Activation)    (None, 20)                0
_________________________________________________________________
dense_2 (Dense)              (None, 1)                 21
_________________________________________________________________
activation_2 (Activation)    (None, 1)                 0
=================================================================
Total params: 61
Trainable params: 61
Non-trainable params: 0
_________________________________________________________________
```

In [22]:

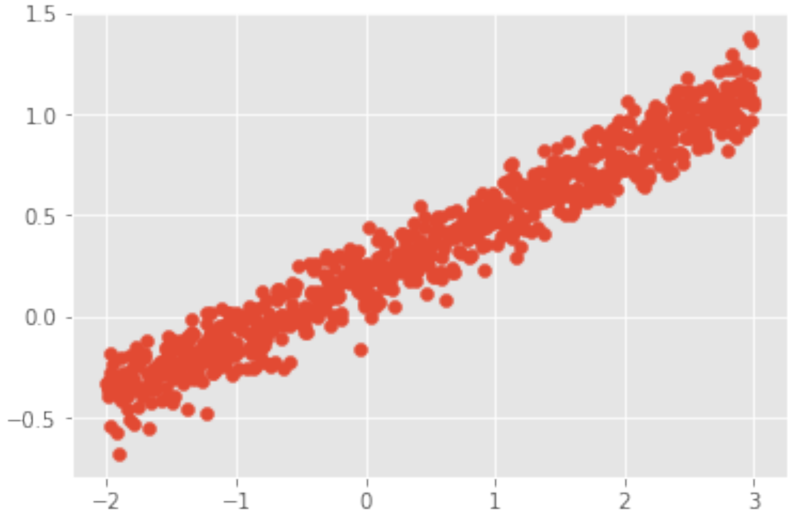
```
#split the data into training and validation set with 80%, 20% resp
ectively
#x_coordinates = np.stack((x, y1), axis=-1)
#y_coordinates = np.stack()
train_x1, val_x1, train_y1, val_y1 = train_test_split(x,y, test_siz
e=0.2, random_state=42,shuffle=True)

print(x.shape)
print(train_x1.shape)
print(y.shape)
print(train_y1.shape)
```

```
(1000,)
(800,)
(1000,)
(800,)
```

Take a look at training dataset

In [23]:
```
plt.scatter(train_x1, train_y1)
```

Out[23]: <matplotlib.collections.PathCollection at 0x145ee0160>



Take a look at validation dataset

In [24]:
```
plt.scatter(val_x1,val_y1)
```

Out[24]: <matplotlib.collections.PathCollection at 0x145f390b8>

In [25]:
```
#OPTIMIZATION PARAMETERS ARE CONFIGURED HERE. THE MODEL IS TRAINED
ALSO
#############################################

#Adam optimize is chosen
#loss function is chosen as mean squared error as it's a regression
problem
#we want also show the results in accurucy results

one_model.compile(optimizer='adam', loss='mean_squared_error',  met
rics=['mae'])

#train the model
history = one_model.fit(train_x1,train_y1,
                    batch_size=128,
                    epochs=10,
                    validation_data=(val_x1, val_y1))
```

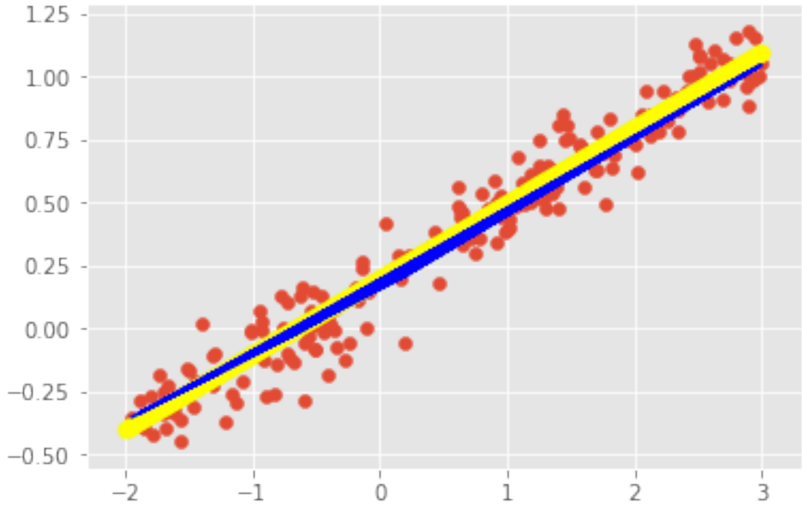
```
W0702 23:45:32.064946 4672939456 deprecation_wrapper.py:119] From
/anaconda3/envs/pytf/lib/python3.6/site-packages/keras/optimizers.
py:790: The name tf.train.Optimizer is deprecated. Please use tf.c
ompat.v1.train.Optimizer instead.

W0702 23:45:32.173138 4672939456 deprecation_wrapper.py:119] From
/anaconda3/envs/pytf/lib/python3.6/site-packages/keras/backend/ten
sorflow_backend.py:986: The name tf.assign_add is deprecated. Plea
se use tf.compat.v1.assign_add instead.

W0702 23:45:32.228817 4672939456 deprecation_wrapper.py:119] From
/anaconda3/envs/pytf/lib/python3.6/site-packages/keras/backend/ten
sorflow_backend.py:973: The name tf.assign is deprecated. Please u
se tf.compat.v1.assign instead.
```

```
Train on 800 samples, validate on 200 samples
Epoch 1/10
800/800 [==============================] - 0s 218us/step - loss: 0
.1589 - mean_absolute_error: 0.3309 - val_loss: 0.1322 - val_mean_
absolute_error: 0.2968
Epoch 2/10
800/800 [==============================] - 0s 12us/step - loss: 0.
1209 - mean_absolute_error: 0.2812 - val_loss: 0.0998 - val_mean_a
bsolute_error: 0.2599
Epoch 3/10
800/800 [==============================] - 0s 9us/step - loss: 0.0
908 - mean_absolute_error: 0.2450 - val_loss: 0.0744 - val_mean_ab
solute_error: 0.2271
Epoch 4/10
800/800 [==============================] - 0s 9us/step - loss: 0.0
676 - mean_absolute_error: 0.2120 - val_loss: 0.0537 - val_mean_ab
solute_error: 0.1931
Epoch 5/10
800/800 [==============================] - 0s 9us/step - loss: 0.0
489 - mean_absolute_error: 0.1796 - val_loss: 0.0382 - val_mean_ab
solute_error: 0.1627
Epoch 6/10
800/800 [==============================] - 0s 9us/step - loss: 0.0
352 - mean_absolute_error: 0.1517 - val_loss: 0.0273 - val_mean_ab
solute_error: 0.1369
Epoch 7/10
800/800 [==============================] - 0s 9us/step - loss: 0.0
257 - mean_absolute_error: 0.1283 - val_loss: 0.0199 - val_mean_ab
solute_error: 0.1157
Epoch 8/10
800/800 [==============================] - 0s 10us/step - loss: 0.
0191 - mean_absolute_error: 0.1107 - val_loss: 0.0153 - val_mean_a
bsolute_error: 0.0997
Epoch 9/10
800/800 [==============================] - 0s 10us/step - loss: 0.
0150 - mean_absolute_error: 0.0976 - val_loss: 0.0126 - val_mean_a
bsolute_error: 0.0883
Epoch 10/10
800/800 [==============================] - 0s 11us/step - loss: 0.
0126 - mean_absolute_error: 0.0893 - val_loss: 0.0111 - val_mean_a
bsolute_error: 0.0830
```

In [26]:
```
pred_y1= one_model.predict(val_x1)
```

In [27]:
```
plt.plot(val_x1,pred_y1,'blue')
plt.scatter(val_x1,val_y1)
plt.scatter(x, 0.3*x+0.2, color = "yellow", linewidth = 2,alpha = .
5)
```

Out[27]: <matplotlib.collections.PathCollection at 0x1468e0438>



Both the model history and the plot above shows that the model performs well. The blue line is created with the validation data set and predicted value on the validation dataset. The red points are noisy validation data. The yellow line is the true equation without adding noise. The blue line and the yellow line almost overlaps, indicating good fitted results.

Problem 4.

Consider three points in (x,y) plane with coordinates (-2,0), (0,1.7) and (2.1,0). Around each of those three centers create a cloud of 100 randomly generated points. For the radial distance of any one of those points from its center use a random normal distribution. For the angular coordinate of any one of “cloud” points use the uniform distribution. Once you have generated all three sets of cloud points plot them in the same diagram using three different colors. There should exist some overlap between the clouds. Create a two-layer neural network. Use Keras API. Fit a model that could predict whether a randomly generated point in the plane belongs to cloud 1, centered around (-2,0), cloud 2, centered around (0,1.7) or cloud 3, centered around (2.1,0). You can make that prediction in a much simpler way, however, we are practicing Keras API.

In [28]:

```
# Generate random data points
random.seed(1024)

#this is the number of points in each cloud
cloud_points=100

#the center of the clouds are:
center1 = [-2, 0]
center2 = [0, 1.7]
center3 = [2.1, 0]

# Sample 100 'r' from normal distribution and 100 'theta' from unif
orm distribution of 0 and 2*pi.
# Then, create 100 points in Cartesian Coordinate System

# radial and angular parameters
mean = 0
sigma = 1.5
low = 0
high = 2*np.pi

#cloud samples for the cloud 1
r1=np.random.normal(mean, sigma, cloud_points)
theta1 = np.random.uniform(low, high, cloud_points)
#cloud samples for the cloud 2
r2=np.random.normal(mean, sigma,cloud_points)
theta2 = np.random.uniform(low,high, cloud_points)
#cloud samples for the cloud 3
r3=np.random.normal(mean, sigma, cloud_points)
theta3 = np.random.uniform(low,high, cloud_points)

# calculate x and y
x1 = center1[0] + r1*np.cos(theta1)
y1 = center1[1]+ r1*np.sin(theta1)
x2 = center2[0] + r2*np.cos(theta2)
y2 = center2[1]+ r2*np.sin(theta2)
x3 = center3[0] + r3*np.cos(theta3)
y3 = center3[1]+ r3*np.sin(theta3)
```

In [29]:
```
#visualize the clouds in 2-dimensional plot
#cloud 1
plt.scatter(x1,y1)
#cloud 2
plt.scatter(x2,y2)
#cloud 3
plt.scatter(x3,y3)


#plot them
plt.show()
```

There're overlapped points indeed, good to go.

In [30]:
```
#class labels for each cloud point (without using one hot encoding)
Y1 = np.tile(np.array([1,0,0]), (100,1))
Y2 = np.tile(np.array([0,1,0]), (100,1))
Y3 = np.tile(np.array([0,0,1]), (100,1))
```

In [31]:
```
# Stack coordinates
X1 = np.stack((x1, y1), axis=-1)
X2 = np.stack((x2, y2), axis=-1)
X3 = np.stack((x3, y3), axis=-1)
```

In [32]:
```
#perform one aggregated input tensor
X=np.concatenate([X1,X2,X3],axis=0)
#perform one aggregated output tensor
Y=np.vstack([Y1,Y2,Y3])
```

In [33]:
```
#split the data into training and validation set with 80%, 20% resp
ectively
train_x, val_x, train_y, val_y = train_test_split( X,Y, test_size=0
.2, random_state=42,shuffle=True)

print(X.shape)
print(train_x.shape)
print(Y.shape)
print(train_y.shape)
```

```
(300, 2)
(240, 2)
(300, 3)
(240, 3)
```

In [34]:
```
# MODEL IS BUILD HERE
# 2 LAYER SIMPLE NEURAL NETWORK
#############################################

#we add the layers in sequential form
model = Sequential()

#first hidden layer with 50 neurons, 2-D coordinates
model.add(Dense(50, input_shape=(2,)))

#relu nonlinear activation function is chosen
model.add(Activation('relu'))

#second hidden layer with 50 neurons
model.add(Dense(50))

#relu nonlinear activation function is chosen
model.add(Activation('relu'))

#output layer - 3 Classes
model.add(Dense(3))

#softmax nonlinear activation function is chosen
model.add(Activation('softmax'))

#show the whole model architecture
model.summary()
############################################
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
dense_3 (Dense)              (None, 50)                150       
_________________________________________________________________
activation_3 (Activation)    (None, 50)                0         
_________________________________________________________________
dense_4 (Dense)              (None, 50)                2550      
_________________________________________________________________
activation_4 (Activation)    (None, 50)                0         
_________________________________________________________________
dense_5 (Dense)              (None, 3)                 153       
_________________________________________________________________
activation_5 (Activation)    (None, 3)                 0         
=================================================================
Total params: 2,853
Trainable params: 2,853
Non-trainable params: 0
_________________________________________________________________
```

In [35]:

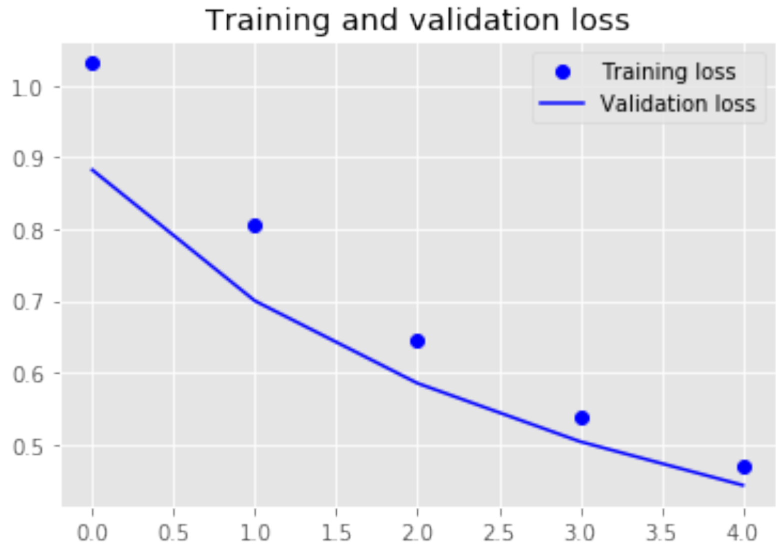
```
#OPTIMIZATION PARAMETERS ARE CONFIGURED HERE. THE MODEL IS TRAINED ALSO
#############################################

#Adam optimize is chosen
#loss function is chosen as categorical_crossentropy
#we want also show the results in accurucy results
model.compile(optimizer=Adam(), loss='categorical_crossentropy', metrics=['accuracy'])

#train the model
history = model.fit(train_x,train_y,
                    batch_size=32,
                    epochs=5,
                    shuffle=True,
                    verbose=1,
                    validation_data=(val_x, val_y))
```

```
W0702 23:45:33.105746 4672939456 deprecation.py:323] From /anacond
a3/envs/pytf/lib/python3.6/site-packages/tensorflow/python/ops/mat
h_grad.py:1250: add_dispatch_support.<locals>.wrapper (from tensor
flow.python.ops.array_ops) is deprecated and will be removed in a
future version.
Instructions for updating:
Use tf.where in 2.0, which has the same broadcast rule as np.where

Train on 240 samples, validate on 60 samples
Epoch 1/5
240/240 [==============================] - 0s 966us/step - loss: 1
.0310 - acc: 0.3875 - val_loss: 0.8823 - val_acc: 0.7833
Epoch 2/5
240/240 [==============================] - 0s 56us/step - loss: 0.
8056 - acc: 0.8042 - val_loss: 0.7000 - val_acc: 0.8833
Epoch 3/5
240/240 [==============================] - 0s 57us/step - loss: 0.
6460 - acc: 0.8625 - val_loss: 0.5853 - val_acc: 0.8833
Epoch 4/5
240/240 [==============================] - 0s 42us/step - loss: 0.
5384 - acc: 0.8625 - val_loss: 0.5041 - val_acc: 0.8500
Epoch 5/5
240/240 [==============================] - 0s 35us/step - loss: 0.
4689 - acc: 0.8708 - val_loss: 0.4434 - val_acc: 0.8500
```

In [36]:

```
#Plot the results for training and validation loss

#history object preserves the loss value
loss = history.history['loss']
#history object preserves the validation loss value
val_loss = history.history['val_loss']

#number of epochs
epochs = range(len(loss))
```

In [37]:
```
plt.figure()

#training loss
plt.plot(epochs, loss, 'bo', label='Training loss')
#validation loss
plt.plot(epochs, val_loss, 'b', label='Validation loss')
#we add title
plt.title('Training and validation loss')

#we add legends
plt.legend()
#finally plot
plt.show()
```



In [38]:
```
# Get predicted probabilities on validationd dataset
model.predict(val_x)
```

Out[38]:
```
array([[0.04382719, 0.03984633, 0.91632646],
       [0.04407236, 0.03922851, 0.9166991 ],
       [0.16286145, 0.6629885 , 0.17415003],
       [0.61444134, 0.22897711, 0.15658163],
       [0.06136416, 0.03273075, 0.9059051 ],
       [0.03478672, 0.03302778, 0.93218553],
       [0.12337618, 0.70124876, 0.17537503],
       [0.206813  , 0.5338764 , 0.25931057],
       [0.670218  , 0.2031522 , 0.12662984],
       [0.15824828, 0.71867055, 0.12308118],
       [0.03291788, 0.33971244, 0.6273697 ],
       [0.15320817, 0.7152921 , 0.13149975],
       [0.11905987, 0.22307512, 0.65786505],
       [0.8564393 , 0.08421239, 0.05934837],
       [0.03651251, 0.9243474 , 0.03914008],
       [0.13077989, 0.45852733, 0.41069272],
       [0.01637244, 0.00283694, 0.9807906 ],
       [0.01024507, 0.00789469, 0.9818602 ],
       [0.11983924, 0.70485574, 0.175305  ],
       [0.11522745, 0.37096068, 0.5138118 ],
       [0.6768374 , 0.2013676 , 0.12179503],
       [0.6369721 , 0.24619654, 0.11683129],
       [0.09082536, 0.09767431, 0.8115003 ],
```

```
       [0.08472509, 0.23206313, 0.68321174],
       [0.06483825, 0.346184  , 0.5889778 ],
       [0.28170654, 0.4823801 , 0.2359134 ],
       [0.2126087 , 0.17020348, 0.61718786],
       [0.04122394, 0.10212123, 0.8566548 ],
       [0.08460816, 0.7815419 , 0.13384993],
       [0.16193658, 0.15006903, 0.68799436],
       [0.6837017 , 0.1955045 , 0.12079389],
       [0.17681386, 0.6540866 , 0.1690995 ],
       [0.69041383, 0.194728  , 0.11485819],
       [0.7318339 , 0.17022632, 0.09793983],
       [0.05974983, 0.07153131, 0.86871886],
       [0.06668438, 0.84767795, 0.08563762],
       [0.89236724, 0.07624731, 0.03138551],
       [0.5263455 , 0.27763486, 0.19601962],
       [0.5920884 , 0.26728496, 0.14062661],
       [0.6772077 , 0.21301821, 0.10977405],
       [0.60557425, 0.2413472 , 0.15307863],
       [0.17913459, 0.55964875, 0.26121664],
       [0.0641022 , 0.06676663, 0.86913115],
       [0.6814613 , 0.20529032, 0.11324834],
       [0.76084906, 0.15666838, 0.08248255],
       [0.7833811 , 0.14883603, 0.06778283],
       [0.10540774, 0.09092611, 0.8036662 ],
       [0.06153851, 0.05256504, 0.8858964 ],
       [0.157631  , 0.6806381 , 0.16173084],
       [0.05614337, 0.05270978, 0.8911469 ],
       [0.21734068, 0.52994365, 0.25271574],
       [0.15037555, 0.41357735, 0.43604717],
       [0.02547319, 0.01442931, 0.9600975 ],
       [0.6756496 , 0.20045125, 0.1238992 ],
       [0.06834172, 0.15476727, 0.776891  ],
       [0.05489345, 0.5169594 , 0.4281471 ],
       [0.6139361 , 0.26718107, 0.11888278],
       [0.68143874, 0.202187  , 0.11637417],
       [0.16109706, 0.63343036, 0.20547263],
       [0.6206221 , 0.24432859, 0.13504937]], dtype=float32)
```

In [39]:

```
# Get predicted classes on validationd dataset

model.predict_classes(val_x)
```

Out[39]:

```
array([2, 2, 1, 0, 2, 2, 1, 1, 0, 1, 2, 1, 2, 0, 1, 1, 2, 2, 1, 2,
0, 0,
       2, 2, 2, 1, 2, 2, 1, 2, 0, 1, 0, 0, 2, 1, 0, 0, 0, 0, 0, 1,
2, 0,
       0, 0, 2, 2, 1, 2, 1, 2, 2, 0, 2, 1, 0, 0, 1, 0])
```

Combined with the probabilities above, cloud 1 is represented by 0, cloud 2 is represented by 1, cloud 3 is represented by 2

In [40]:
```
# Generate new data in cloud 1, around center 1
random.seed(1023)
#cloud samples for the cloud 1
r1_new=np.random.normal(mean, sigma, cloud_points)
theta1_new = np.random.uniform(low, high, cloud_points)
# calculate x and y
x1_new = center1[0] + r1_new*np.cos(theta1)
y1_new = center1[1]+ r1_new*np.sin(theta1)
X1_new = np.stack((x1_new, y1_new), axis=-1)
```

In [41]:
```
model.predict_classes(X1_new)
```

Out[41]:
```
array([0, 0, 2, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0,
0, 0,
       0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 0, 1, 0,
0, 2,
       1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0,
       0, 1, 2, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1,
0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2])
```

The result on the new data is pretty good as most of the entries are 0 or cloud 1.