

Applied Machine Learning: Homework #2

Due on DUE DATE: OCT 23 11:59 PM

CLASS INSTRUCTOR: Volodymyr Kuleshov

Xingyue Dai(xd86) Wentao Xu(wx225)

PROGRAMMING EXERCISES

Problem 1

Binary Classification on Text Data. In this problem you will use several machine learning techniques from the class to perform classification on text data. Throughout the problem, we will be working on the NLP with Disaster Tweets Kaggle competition, where the task is to predict whether or not a tweet is about a real disaster.

Solution

- (a) There are 7613 training data points and 3263 test data points.

There are 3271 real tweets, which is 42.9%. There are 4342 not real tweets, which is 57%.

codes:

```
def read_data(path):
    '''
    read nlp data
    '''
    file = open(path, "r")
    reader = csv.reader(file)
    data = []
    for line in reader:
        if reader.line_num == 1:
            continue
        data.append(line)
    return data

def stat_train_test_data_point(train, test):
    train_point = len(train)
    test_point = len(test)
    print(train_point)
    print(test_point)

def stat_real_disasters(data):
    true = 0
    false = 0
    for i in data:
        if i[4] == '1':
            true += 1
        else:
            false += 1
    print(true)
    print(false)
```

- (b) codes:

```
def split_train(data):
    ratio = 0.7
    train_set = []
    len_train = 0
```

```

len_total = len(data)
indexes = []
removed_set = set()
while len_train < int(0.7*len_total):
    index = int(random.random() * len(data))
    while (index in removed_set):
        index = int(random.random() * len(data))
    removed_set.add(index)
    indexes.append(data[index])
    len_train += 1
for i in indexes:
    train_set.append(i)
    data.remove(i)
dev_set = np.array(data)
train_set = np.array(train_set)
dev = {"id":dev_set[:,0].tolist(),
       "keyword":dev_set[:,1].tolist(),
       "location":dev_set[:,2].tolist(),
       "text":dev_set[:,3].tolist(),
       "target":dev_set[:,4].tolist()}
train = {"id":train_set[:,0].tolist(),
         "keyword":train_set[:,1].tolist(),
         "location":train_set[:,2].tolist(),
         "text":train_set[:,3].tolist(),
         "target":train_set[:,4].tolist()}
path = "./nlp-getting-started/"
t = DataFrame(train)
d = DataFrame(dev)
d.to_csv(path + "dev_set.csv", index=False)
t.to_csv(path + "train_set.csv", index=False)

```

- (c) I converted all characters to lowercase, lemmatize all the words, strip punctuation and strip the stop words, most of which are based on nltk package.

The reason why I do this is that we need to format all the same words which have uppercase characters into the same lowercase mode, also, I format the 'adj', 'verb' and 'noun' into their regular mode, which helps computer to calculate the possibilities.

I also delete punctuations and stop words, which will confuse the calculation process of our models. codes:

```

def preprocess_data(data):
    lemmatizer = WordNetLemmatizer()
    tokenizer = RegexpTokenizer(r'\w+')
    punc = string.punctuation
    stop_words = set(stopwords.words('english'))
    for i in range(len(data)):
        data[i][3] = data[i][3].lower()
        # data[i][3] = re.split(' ', data[i][3])
        # data[i][3] = tokenizer.tokenize(data[i][3])
        data[i][3] = word_tokenize(data[i][3])
    delete_list = []

```

```

for j, word in enumerate(data[i][3]):
    if word.startswith("//t"):
        delete_list.append(word)
    elif word.startswith("http"):
        delete_list.append(word)
    elif (word in punc):
        delete_list.append(word)
    elif (word.isalpha() == False):
        delete_list.append(word)
    elif (word in stop_words):
        delete_list.append(word)
for word in delete_list:
    data[i][3].remove(word)
for j, word in enumerate(data[i][3]):
    processed_word = lemmatizer.lemmatize(word, pos="v")
    processed_word = lemmatizer.lemmatize(processed_word, pos="a")
    processed_word = lemmatizer.lemmatize(processed_word, pos="n")
    data[i][3][j] = processed_word
return data

```

- (d) Here I think the value of min_df depends on models, since different models may be sensible to different min_df values, so here I just set it as a parameter. Also I made both word bags for train and dev sets. Here I set min_df as 1 as the example, the total number of features in these vectors are 9953 when the min_df is 1.

```

def word_bags(data, mindf):
    # decide the threshold M
    words = []
    words_set = set()
    label = []
    for i in range(len(data)):
        temp = ""
        for j in range(len(data[i][3])):
            temp += data[i][3][j]
            words_set.add(data[i][3][j])
            if j != len(data[i][3])-1:
                temp += " "
        words.append(temp)
        label.append(int(data[i][4]))
    count_vect = CountVectorizer(binary=True, min_df=mindf)
    X_train = count_vect.fit_transform(words)
    label = np.array(label)
    return count_vect, X_train, label

```

- (e) Here I implemented naive Bayes classifier based on codes on the lec8.

```

def merge_data(data):
    label = []
    words = []
    for i in range(len(data)):

```

```

        temp = ""
        for j in range(len(data[i][3])):
            if j!=len(data[i][3])-1:
                temp = temp + data[i][3][j] + " "
            else:
                temp = temp + data[i][3][j]
        words.append(temp)
        label.append(int(data[i][4]))
    return words, label

def naive_bayes(data, train, dev):
    # decide the threshold M
    data = train
    count_vect, X_train, label = word_bags(data, 3)
    n = X_train.shape[0]
    d = X_train.shape[1]
    K = 2
    # shapes of parameters
    psis = np.zeros([K, d])
    phis = np.zeros([K])
    # compute the parameters
    for k in range(K):
        X_k = X_train[label == k]
        psis[k] = np.mean(X_k, axis=0)
        phis[k] = X_k.shape[0] / float(n)
    dev_data, dev_label = merge_data(dev)
    dev_data = count_vect.transform(dev_data).toarray()
    def nb_predictions(data, psis, phis):
        x = data
        n, d = x.shape
        x = np.reshape(x, (1, n, d))
        psis = np.reshape(psis, (K, 1, d))
        # clip probabilities to avoid log(0)
        psis = psis.clip(1e-14, 1-1e-14)
        # compute log-probabilities
        logpy = np.log(phis).reshape([K,1])
        logpxy = x * np.log(psis) + (1-x) * np.log(1-psis)
        logpyx = logpxy.sum(axis=2) + logpy
        return logpyx.argmax(axis=0).flatten(), logpyx.reshape([K,n])
    idx, logpyx = nb_predictions(dev_data, psis, phis)
    TP = 0
    FP = 0
    FN = 0
    TN = 0
    for i, j in zip(idx, dev_label):
        if i and j:
            TP += 1
        elif i==1 and j!=1:
            FP += 1

```

```

        elif i!=1 and j==1:
            FN += 1
        else:
            TN += 1
    precision = TP / (TP + FP)
    recall = TP / (TP + FN)
    F1 = 2*(precision * recall) / (precision + recall)
    print(F1)
    return F1

```

The F1 score I got is 0.754 under the condition where the value of min_df is 3, which is the best F1 score.

(f) Here I found when mindf = 1, it can generate best model.

```

def get_weight(count_vect, model, num):
    # get important weight
    coef = model.coef_
    coef = [abs(i) for i in coef[0]]
    max_num_index_list = map(coef.index, heapq.nlargest(num, coef))
    max_num_index_list = list(max_num_index_list)
    import_words = set()
    most_ = max_num_index_list
    for i in count_vect.vocabulary_.keys():
        if count_vect.vocabulary_[i] in most_:
            import_words.add(i)
    influential_words = []
    for i in import_words:
        influential_words.append(i)
    return influential_words

def logistic(data, train, dev):
    # decide the threshold M
    data = train
    count_vect, X_train, label = word_bags(data, 1)
    dev_data, dev_label = merge_data(dev)
    dev_data = count_vect.transform(dev_data).toarray()
    logreg = LogisticRegression(C=1e5, multi_class='multinomial', verbose=True)
    logreg.fit(X_train, label)
    dev_predict = logreg.predict(dev_data)
    F1 = f1_score(dev_label, dev_predict)
    print(F1)
    # get important weight
    influential_words = get_weight(count_vect, logreg, 10)
    print(influential_words)
    return influential_words, F1

```

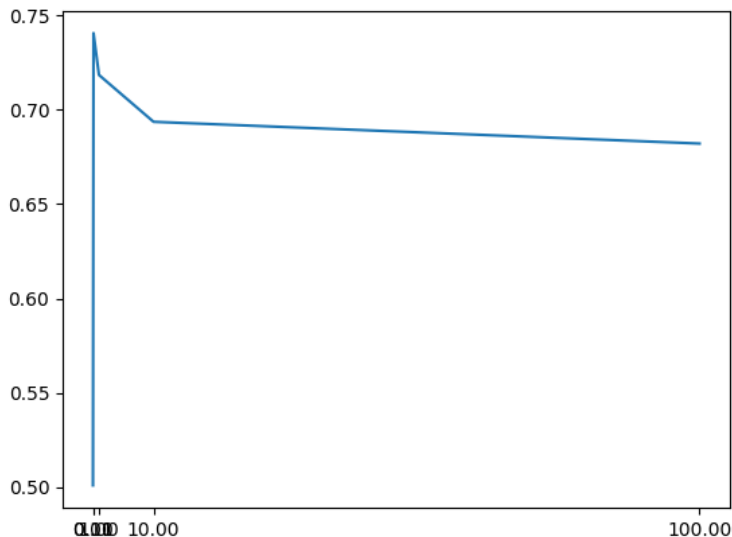
The F1 score I get is 0.7034, and I choose ten important weight and the corresponding words:

['wildfire', 'suicide', 'hiroshima', 'derailment', 'better', 'massacre', 'volcano', 'curfew', 'horrible', 'hail-storm']

```
(g) def linear_svm(data, train, dev):
    # decide the threshold M
    data = train
    count_vect, X_train, label = word_bags(data, 1)
    dev_data, dev_label = merge_data(dev)
    dev_data = count_vect.transform(dev_data).toarray()
    clf = LinearSVC(penalty='l2', loss='hinge', dual=True, C=0.1)
    clf.fit(X_train, label)
    dev_predict = clf.predict(dev_data)
    F1 = f1_score(dev_label, dev_predict)
    print(F1)
    # get important weight
    influential_words = get_weight(count_vect, clf, 10)
    print(influential_words)
    return influential_words, F1
```

I tried different C values, and plot a graph.

```
def C_plot(total_processed_data, processed_data, processed_dev, C):
    F = []
    for i in C:
        a, b = linear_svm(total_processed_data, processed_data, processed_dev, i)
        F.append(b)
    plt.plot([0.01, 0.1, 1, 10, 100], F)
    plt.xticks([0.01, 0.1, 1, 10, 100])
    plt.savefig("./linear_svm.png")
```



and I found the best model can be generated when $C = 0.1$, where F1 is 0.74057. The best model here compared with the logistic regression classifier performs better, which generates higher F1 value.

And the important words I get is

['drought', 'massacre', 'suicide', 'earthquake', 'flood', 'storm', 'fire', 'deaths', 'hiroshima', 'wildfire']

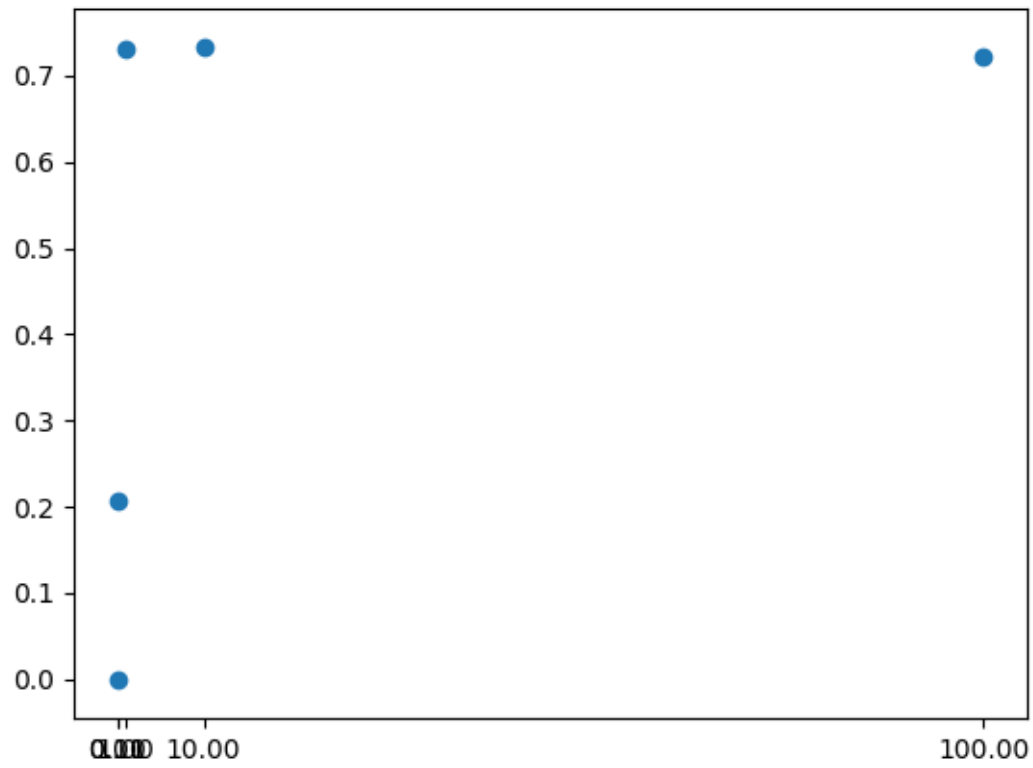
which are similar but not the same as the important words in logistic regression models.

(h) Here I used Gaussian kernel.

```
def non_linear_svm(data, train, dev, c):
    # decide the threshold M
    data = train
    count_vect, X_train, label = word_bags(data, 1)
    dev_data, dev_label = merge_data(dev)
    dev_data = count_vect.transform(dev_data).toarray()
    clf = SVC(C=c, kernel="rbf")
    clf.fit(X_train, label)
    dev_predict = clf.predict(dev_data)
    F1 = f1_score(dev_label, dev_predict)
    print(F1)
    return F1

def non_lin_C_plot(total_processed_data, processed_data, processed_dev, C):
    F = []
    for i in C:
        b = non_linear_svm(total_processed_data, processed_data, processed_dev, i)
        F.append(b)
    print(F)
    plt.plot([0.01, 0.1, 1, 10, 100], F)
    plt.xticks([0.01, 0.1, 1, 10, 100])
    plt.savefig("./non_linear_svm.png")
```

The graph:



So we can see that 10 is the best C value, where the F1 score is 0.733.

- (i) Here in this problem, I just post some key codes which I used to generate n-gram, other codes will be attached and submitted through code parts.

I set threshold to be 1 since I found that if I set it to be larger than 1, models will generate poor results. The total number of 1-grams and 2-grams in my vocabulary is 40648.

I randomly choose 10 2-grams:

utc volcano
 volcano hawaii
 loan upheaval
 upheaval way
 way oneself
 oneself save
 save house
 house leave
 leave foreclose

Below are codes:

```
def n_gram(data, mindf):
    # decide the threshold M
```

```

words = []
words_set = set()
label = []
for i in range(len(data)):
    temp = ""
    for j in range(len(data[i][3])):
        temp += data[i][3][j]
        words_set.add(data[i][3][j])
        if j != len(data[i][3])-1:
            temp += " "
    words.append(temp)
    label.append(int(data[i][4]))
count_vect = CountVectorizer(binary=True, min_df=min_df, ngram_range=(1,2))
X_train = count_vect.fit_transform(words)

X_show = X_train.toarray()
print(X_show.shape[0])
print(X_show.shape[1])
print(len(count_vect.vocabulary_))

two_gram_vec = CountVectorizer(binary=True, min_df=min_df, ngram_range=(2,2))
two_gram_vec.fit_transform(words)
countt = 0
# print(type(two_gram_vec.vocabulary_))
for i in two_gram_vec.vocabulary_.keys():
    print(i)
    countt += 1
    if countt == 9:
        break
label = np.array(label)
return count_vect, X_train, label

```

For (e), the F1 value is 0.73, and I set min_df to be 1.

For (f), the F1 value is 0.726, and the important words returned are:

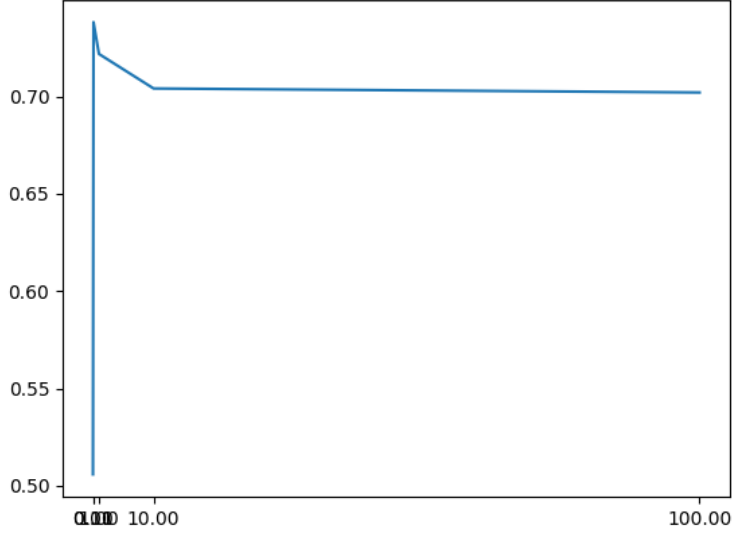
['kill', 'california', 'massacre', 'curfew', 'hiroshima', 'jorrynja', 'deaths', 'suicide', 'wildfire']

Here I set the min_df to be 1.

For (g), the best F1 is 0.738, when C is 0.1 and min_df is 1, important words are:

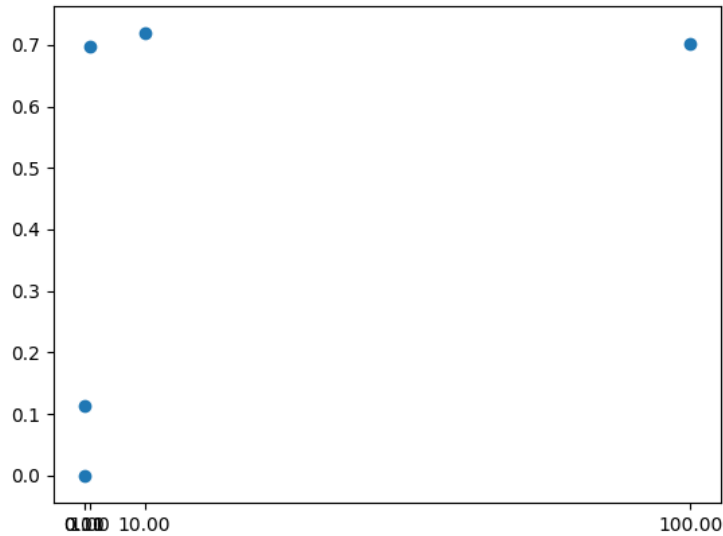
['fire', 'flood', 'storm', 'wildfire', 'hiroshima', 'earthquake', 'drought', 'volcano', 'deaths', 'massacre']

And below is the plot:



For (h), the best F1 is 0.719, when C is 10, and min_df is 1.

Below is the plot:



The above results show that there are some differences but not many differences from those using the bag of word model. But some models' F1 values are raised.

I think it is because that some models take consideration on the relationship of different words, which also provide some information useful to classifying tasks in some models (not all models).

- (j) Here, I just try appending words in keyword column and location column into text column and recalculate the bag of words, and I tried bag of words model.

The reason I choose appending words in keyword column and location column is that I think the locations and keywords of same labeled data may have some common information, and at the same

time, they can be regarded as meaningful words which are the same as the words in the text columns, so I just append them and then do the bag of words.

so I choose to do this, and the results show that some models' F1 value do have been raised.

And word of bag models perform well in the above problems, so I choose word of bags.

Below are codes:

```
def append_data(data):
    for i in range(len(data)):
        temp_str = ""
        temp_str = temp_str + data[i][1] + " "
        temp_str = temp_str + data[i][2] + " "
        temp_str = temp_str + data[i][3]
        data[i][3] = temp_str
    return data
```

For (f), the min_df I set is 1, while the result is not very good, the best F1 is just 0.694. Below are important words:

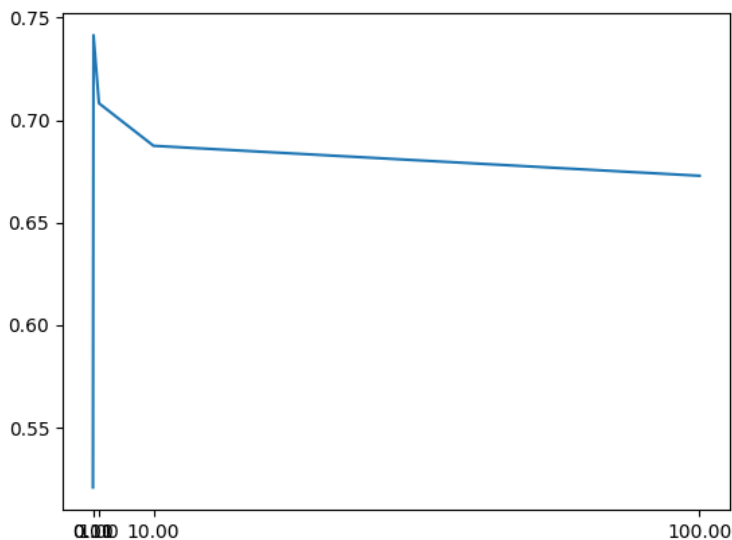
['better', 'volcano', 'blight', 'outbreak', 'hiroshima', 'derailment', 'typhoon', 'curfew', 'wildfire', 'suicide']

The F1 values are decreased, I think it is because the logistic regression model is not sensible to these locations and key words, and locations and key words make the computation difficult.

For (g), the best F1 score is 0.741, where I set mind_df to be 1, the best C is 0.1. The important words are:

['storm', 'wildfire', 'train', 'deaths', 'suicide', 'drought', 'earthquake', 'fire', 'hiroshima', 'kill']

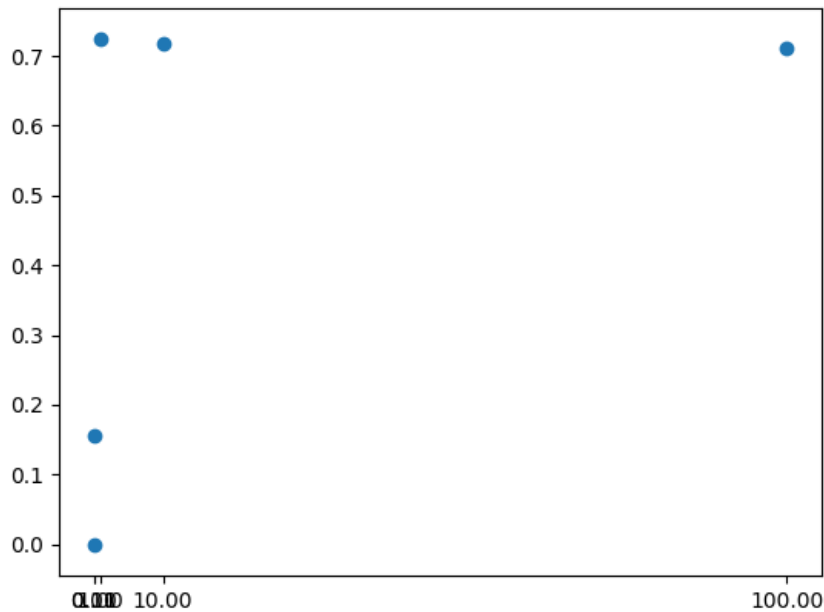
Below is graph:



The F1 value has been slightly raised, which means locations and key words help linear SVM to divide data sets.

For (h), the best F1 score is 0.724, where I set mind_df to be 1, the best C is 1.

Below is graph:



The F1 value of non-linear SVM has been slightly decreased, which means that for models which have some non-linear properties (including logistic regression models), locations and key words may not be a very good choice.

- (k) So the final model I used is naive Bayes classifier of word bag model, which returned me the largest F1 score, 0.75. And I train it on the entire Kaggle train data and test it on the Kaggle test data.

Below are codes:



```
def predict_test(train, dev):
    data = train
    id = []
    for i in range(len(dev)):
        id.append(dev[i][0])
    count_vect, X_train, label = word_bags(data, 3)
    n = X_train.shape[0]
    d = X_train.shape[1]
    K = 2
    # shapes of parameters
    psis = np.zeros([K, d])
    phis = np.zeros([K])
    # compute the parameters
    for k in range(K):
        X_k = X_train[label == k]
        psis[k] = np.mean(X_k, axis=0)
        phis[k] = X_k.shape[0] / float(n)
    dev_data = trans(dev, count_vect)
    dev_data = count_vect.transform(dev_data).toarray()
```

```

def nb_predictions(data, psis, phis):
    x = data
    n, d = x.shape
    x = np.reshape(x, (1, n, d))
    psis = np.reshape(psis, (K, 1, d))
    # clip probabilities to avoid log(0)
    psis = psis.clip(1e-14, 1-1e-14)
    # compute log-probabilities
    logpy = np.log(phis).reshape([K,1])
    logpxy = x * np.log(psis) + (1-x) * np.log(1-psis)
    logpyx = logpxy.sum(axis=2) + logpy
    return logpyx.argmax(axis=0).flatten(), logpyx.reshape([K,n])
idx, logpyx = nb_predictions(dev_data, psis, phis)
ex = {"id":idx,
      "target":idx}
d=DataFrame(ex)
d.to_csv("./submit.csv", index=False)

```

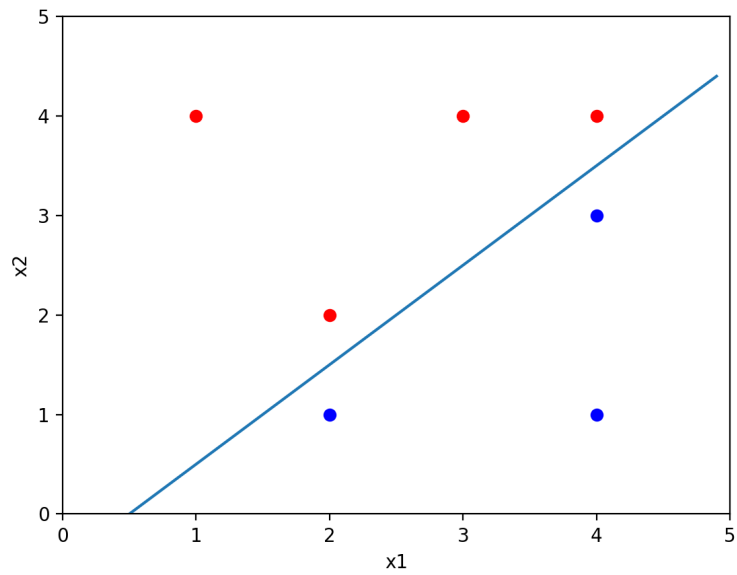
Kaggle returned us the result:

750	WentXu		0.78639	2	4h
Your Best Entry 					
Your submission scored 0.78210, which is not an improvement of your best score. Keep trying!					

Which is higher than my expectation, I think there are lots of reasons, maybe the information or the features of test_data are more similar with the train_sets than dev sets.

- (1) I would like to choose a SVM model instead for a consulting job. The naive Bayes involves the statistics knowledge and an assumption that all variables in the naive Bayes model are independent, which generates the final naive Bayes classifier. The above two key points of naive Bayes classifier may not convincing my clients, the rationality of naive Bayes is not as good as its performance.

So I would like choose SVM and use a 2-D plot to convince my clients. The example 2-D plot I used are as below:



SVM can select boundary with high margin, we are as confident as possible for every point and we are as far as possible from the decision boundary for separable data set.

When it comes to non-separable data set, we can use kernel skills to make them be separable data set again, which I think can convince my clients.

WRITTEN EXERCISES

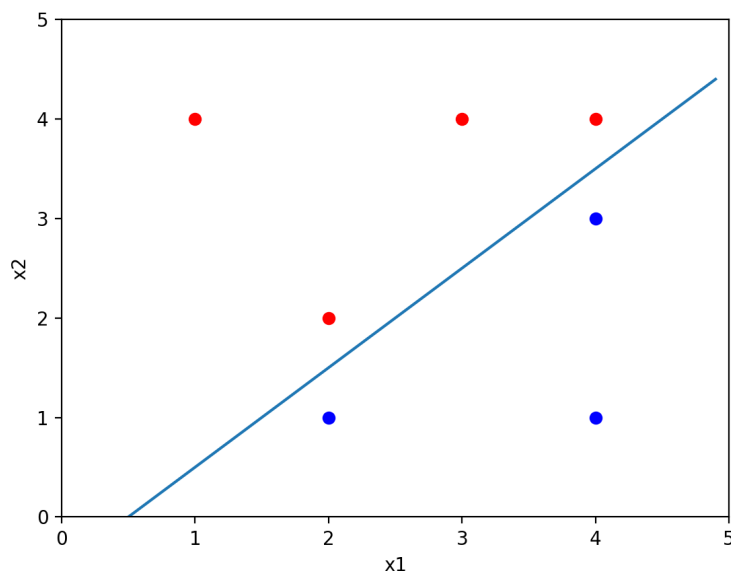
Problem 2

Maximum-margin classifiers. Suppose we are given $n = 7$ observations in $p = 2$ dimensions. For each observation, there is an associated class label.

- (a) Sketch the observations and the maximum-margin separating hyperplane.
- (b) Describe the classification rule for the maximal margin classifier. It should be something along the lines of “Classify to Red if $\beta_0 + \beta_1 X_1 + \beta_2 X_2 \geq 0$, and classify to Blue otherwise.” Provide the values for β_0, β_1 , and β_2 .
- (c) On your sketch, indicate the margin for the maximal margin hyperplane.
- (d) Indicate the support vectors for the maximal margin classifier.
- (e) Argue that a slight movement of the seventh observation would not affect the maximal margin hyperplane.
- (f) Sketch a hyper plane that separates the data, but is not the maximum-margin separating hyperplane. Provide the equation for this hyperplane.
- (g) Draw an additional observation on the plot so that the two classes are no longer separable by a hyperplane.

Solution

- (a) Below are the observations and the maximum-margin hyperplane



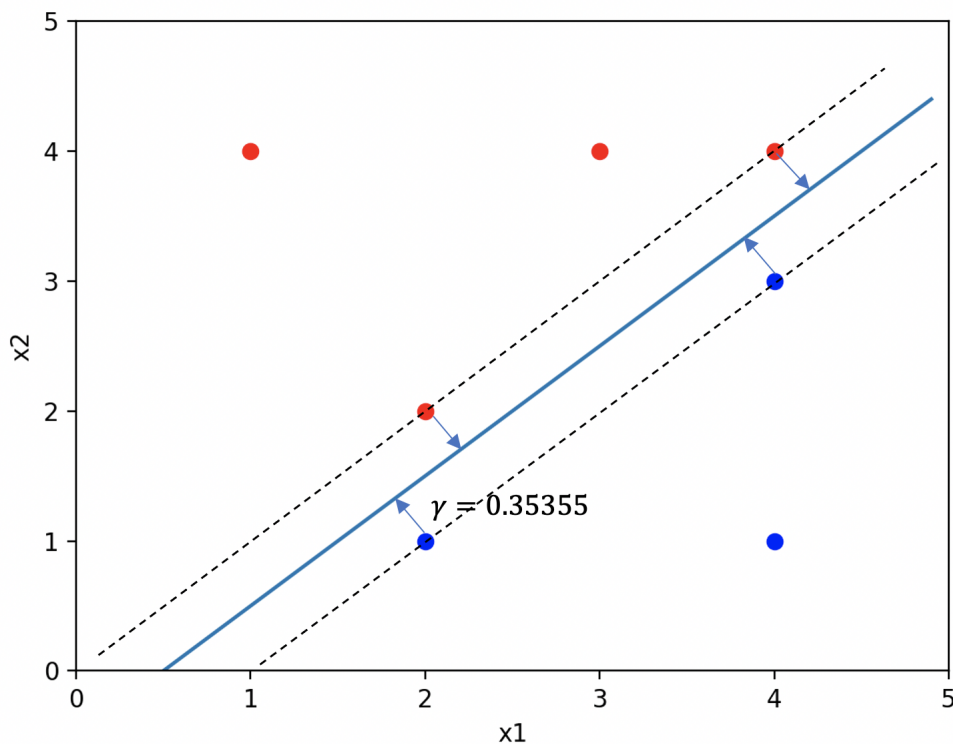
- (b)

The maximum margin hyperplane has the formula $-0.5 + x_1 - x_2 = 0$

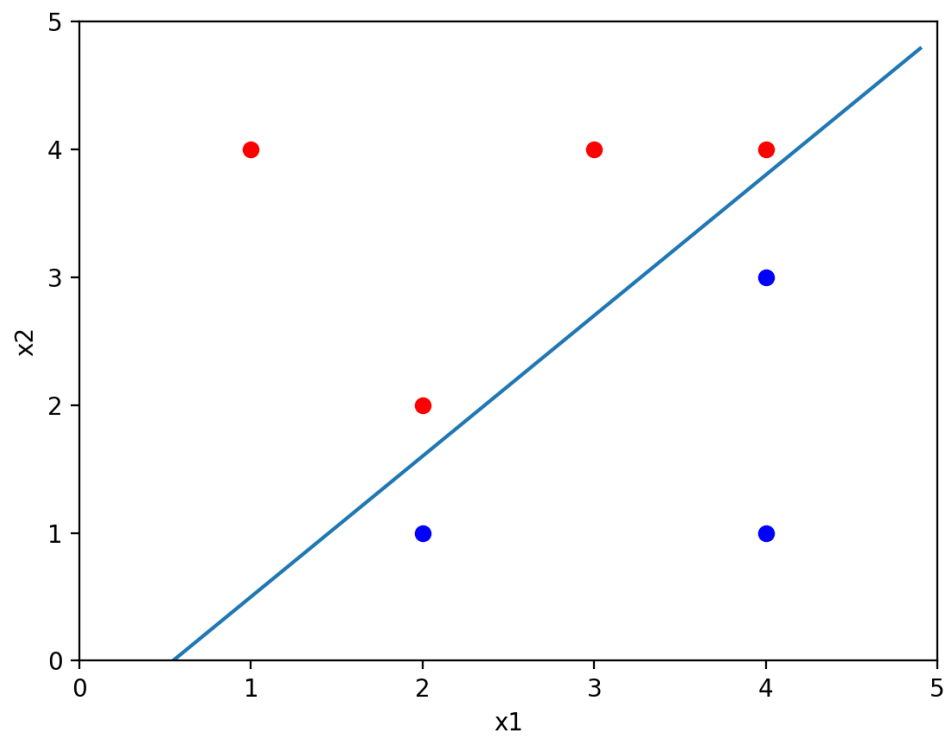
Thus, we have $\beta_0 = -0.5$, $\beta_1 = 1$ and $\beta_2 = -1$

if $-0.5 + x_1 - x_2 > 0$, then the data points should be classified as 'Blue', otherwise they should be classified as 'Red'.

- (c) take point(2,1) as a pivot, we then have the margin $\gamma = (-0.5 + 2 - 1)/\sqrt{1 + 1} = 0.35355$ as shown below

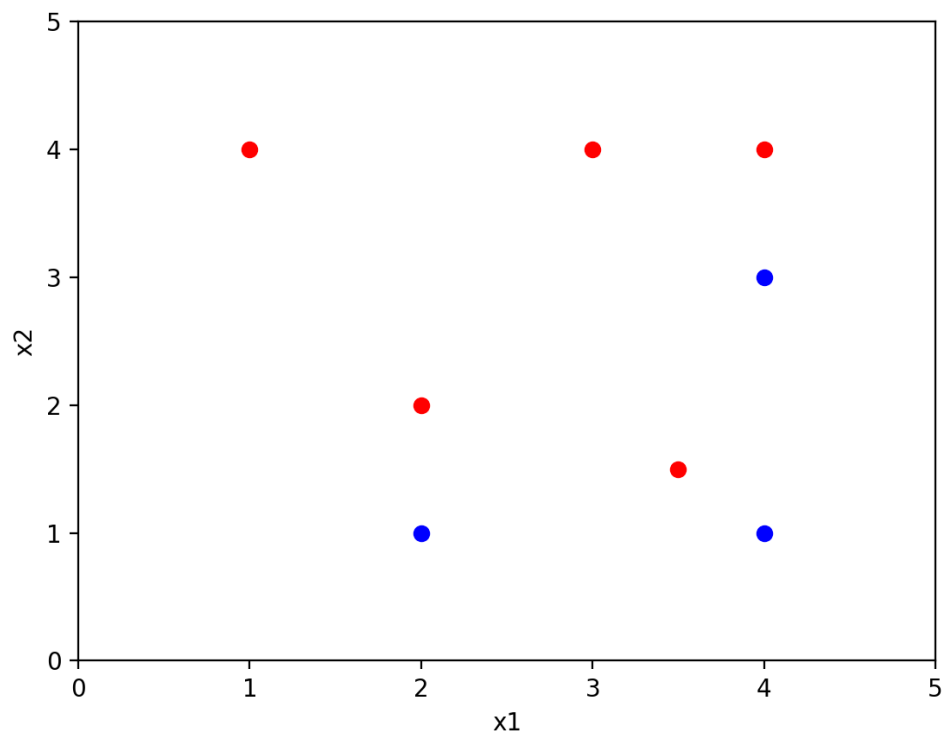


- (d) the support vectors are (2,1), (2,2), (4,3) and (4,4) as they lie closest to the maximum-margin hyperplane with $\gamma = 0.35355$
- (e) The seventh observation is (4,1) which lies very far from the maximum-margin hyperplane. Since such a hyperplane is determined and only determined by support vectors, a slight movement of a far-away vector like (4,1) would not affect the maximal-margin hyperplane.
- (f) As shown below, another hyperplane separates the data with the equation $-0.6 + 1.1x_1 - x_2 = 0$



(g)

below is an example that with this extra observation $(3.5, 1.5)$, this data is no longer separable by a hyperplane



WRITTEN EXERCISES

Problem 3

Naive Bayes with binary features. You are working at a hospital, and tasked with developing a classifier to predict whether patients of some hospital have the flu or not based on their symptoms. Suppose that you have access to the following information about the distribution of patients entering the hospital:

- 20% of the patients have the flu, and 80% do not
- Out of the patients who have the flu:
 - 75% have both coughing and sneezing
 - 5% have sneezing but no coughing
 - 5% have coughing but no sneezing
 - 15% have neither coughing nor sneezing
- Out of the patients who do not have the flu:
 - 4% have both coughing and sneezing
 - 1% have sneezing but no coughing
 - 1% have coughing but no sneezing
 - 94% have neither coughing nor sneezing

- Suppose the hospital is presented with a patient who has both coughing and sneezing. According to the full generative model presented above, what is the probability that the patient doesn't have the flu?
- Now, suppose we wish to train a naive Bayes classifier using the above data. What is the probability that the above patient doesn't have the flu according to a naive Bayes model?
- Do the above approaches give significantly different probabilities that the above patient doesn't have the flu? If so, why? Which approach do you think gives a more reasonable estimate? You should relate your answer to the different assumptions made by the two approaches.

Solution

- Let F, C, S be the random variables that people have flu, coughing and sneezing respectively. Then we have

$$P(F) = 0.2, P(F^c) = 1 - 0.2 = 0.8$$

$$P(C, S | F) = 0.75, P(S, C^c | F) = 0.05, P(C, S^c | F) = 0.05, P(C^c, S^c | F) = 0.15,$$

$$P(C, S | F^c) = 0.04, P(S, C^c | F^c) = 0.01, P(C, S^c | F^c) = 0.01, P(C^c, S^c | F^c) = 0.94$$

Then,

$$P(F^c | C, S) = \frac{P(C, S | F^c)P(F^c)}{P(C, S)} = \frac{P(C, S | F^c)P(F^c)}{P(C, S | F)P(F) + P(C, S | F^c)P(F^c)} = \frac{0.04 * 0.8}{0.75 * 0.2 + 0.04 * 0.8} = 0.1758$$

- According to the Naive Bayes model which factorizes the probability over each dimension, we have

$$\begin{aligned} P(F^c | C, S) &= \frac{P(C, S | F^c)P(F^c)}{P(C, S)} = \frac{P(C | F^c)P(S | F^c)P(F^c)}{P(C | F)P(S | F)P(F) + P(C | F^c)P(S | F^c)P(F^c)} \\ &= \frac{(P(C, S^c | F^c) + P(C, S | F^c))(P(S, C | F^c) + P(S, C^c | F^c))P(F^c)}{(P(C, S^c | F) + P(C, S | F))(P(S, C | F) + P(S, C^c | F))P(F) + (P(C, S^c | F^c) + P(C, S | F^c))(P(S, C | F^c) + P(S, C^c | F^c))P(F^c)} \\ &= \frac{(0.01 + 0.04)(0.04 + 0.01)0.8}{(0.05 + 0.75)(0.75 + 0.05)0.2 + (0.01 + 0.04)(0.04 + 0.01)0.8} = \frac{0.002}{0.128 + 0.002} = 0.01538 \end{aligned}$$

- Yes, the probabilities generated from these two approaches are very different. Because the first approach considers people who have symptoms of coughing and sneezing while not catching a flu among all people. While the second approach assumes that we're given a specific person who has symptoms of coughing and sneezing, and we want to know the probability that he doesn't have a flu, in which case

we have a smaller sample space. Therefore, in the event "the hospital is presented with a patient", the second approach with naive Bayes gives a more reasonable estimate.

WRITTEN EXERCISES

Problem 4

Solution

(a) According to the question title, we have

$$\theta_{t+1} = \theta_t - \alpha \left(\frac{1}{n} \sum_{i=1}^n (y_i - \theta_t^T \phi(x_i)) \phi(x_i) + \lambda \theta_t \right) = \theta_t - \frac{\alpha}{n} \sum_{i=1}^n (y_i - \theta_t^T \phi(x_i)) \phi(x_i) - \alpha \lambda \theta_t$$

let $\theta_t = \sum_{i=1}^n \beta_i^{(t)} \phi(x_i)$, then we have

$$\theta_{t+1} = \sum_{i=1}^n \beta_i^{(t)} \phi(x_i) - \frac{\alpha}{n} \sum_{i=1}^n (y_i - \sum_{j=1}^n \beta_j^{(t)} \phi(x_j)^T \phi(x_i)) \phi(x_i) - \alpha \lambda \sum_{i=1}^n \beta_i^{(t)} \phi(x_i)$$

Let $L_{i,j} = \phi(x_i)^T \phi(x_j)$, then we have

$$\theta_{t+1} = \sum_{i=1}^n ((1 - \alpha \lambda) \beta_i^{(t)} - \frac{\alpha}{n} y_i + \frac{\alpha}{n} \sum_{j=1}^n \beta_j^{(t)} L_{j,i}) \phi(x_i)$$

which then shows that

$$\beta_i^{(t+1)} = (1 - \alpha \lambda) \beta_i^{(t)} - \frac{\alpha}{n} y_i + \frac{\alpha}{n} \sum_{j=1}^n \beta_j^{(t)} L_{j,i}$$

which, in a compact form, shows that

$$\beta^{(t+1)} = (1 - \alpha \lambda) \beta^{(t)} - \frac{\alpha}{n} y + \frac{\alpha}{n} \beta^{(t)} L$$

(b) $\hat{y} = \theta_t^T \phi(x_{test}) = \sum_{i=1}^n \beta_i^{(t)} \phi(x_i)^T \phi(x_{test}) = \sum_{i=1}^n \beta_i^{(t)} k(x_i, x_{test})$

Where $k(x, x') = \phi(x)^T \phi(x')$

(c) Because $\phi(x)$ are never used directly in both equations generated from part(a) and part(b). Only the dot product of $\phi(x)$ which is the kernel function K is used in the previous two equations. And according to the question title, K is easy to compute. So, we will be able to perform gradient descent and computing predicted values even if $\phi(x)$ is infinite-dimensional by using the kernel trick to compute k efficiently without tacking $\phi(x)$ itself.