OREGON STATE UNIVERSITY

CS534: Machine Learning
Homework 2

Shangjia Dong, Chuan Tian

December 20, 2017

0 STARTING POINT (FROM HW1)

You can re-use your HW1 code on feature maps. Let's start with the very basic fully binarized feature map, which resulted in 231 observed features (including the bias dimension) on income.train.txt.5k. Running perceptron and Aggressive MIRA (0.9) for 5 epochs on these 5,000 examples (with everything else also being default) should result in these train/dev error rates (here only showing the averaged versions):

perceptron avg: train_err 16.16% (+:20.3%)	dev_err 16.11% (+:19.5%) at epoch 3.40
AMIRA 0.9 avg: train_err 16.38% (+:21.0%)	dev_err 16.05% (+:18.9%) at epoch 0.60

I confirmed that the error rates from the unaveraged versions are slightly worse than those trained on the full training set in HW1, but the averaged versions are almost identical (which is reasonable, given that both averaged perceptron/MIRA achieved their best error rates before a full epoch on HW1). So we will use the smaller 5k training set instead of the whole HW1 set for faster SVM training. If your numbers are too far from these, you may use my reference HW1 code (on course homepage)

1 BASIC (LINEAR) SVM FROM SKLEARN

1. Train SVM using sklearn.svm.SVC on the 5k training set, using linear kernel (kernel='linear') and default the C = 1. What's your training and dev error rates?
Note: the train error rate should be better than averaged perceptron/AMIRA, and the dev error rate should be similar to averaged AMIRA.

Note: this training might take a while depending on the computer. It took about 4 secs on my laptop (2015 MacBook Pro) and about 3 secs on pelican machines but about 22 secs on flip machines (default ENGR machines if you ssh access.engr.oregonstate.edu)! These flip machines are not meant for computing jobs; use pelican instead (ssh pelican.eecs.oregonstate.edu) if you want.
Also report the time it takes to train
Answer:

Table 1.1: Model Performance

	Training error (%)	Validation error (%)	Training time (sec)
Linear SVM	15.72	16.05	3.158
Average Perceptron	16.16	16.11	1.498
AMIRA	16.26	16.05	1.583

2. How many support vectors are there? (clf.n_support_) How many among them have margin violations (i.e., non-zero slacks)?
Answer: There are 1964 support vectors in total ([1005, 959]). To calculate the margin violation, we used two criteria: (1) slack > 0; (2) α = C. There are 1791 instances have margin violation. However, if we only use the criteria of slack > 0, then we have 1869 instances. It doesn't make sense to have vectors which are margin violations but not have α = C. We discussed with classmates and concluded that it might be due to float-point accuracy. So our final answer is still 1791 instances.

3. Calculate the total amount of margin violations (i.e., slacks): $\sum_i \xi$ and the objective $\frac{1}{2}w^2 + C\sum_i \xi_i$.
Answer: Summarize all the margin violations in the training set, we have margin violation 1802.33 in total. The objective function is calculated as 1840.15.

4. For both (positive, negative) classes, list the top five most violated training examples and their slacks.
Answer:
Positive:
['54' 'Self-emp-not-inc' 'Some-college' 'Divorced' 'Farming-fishing' ' White' 'Female' '30' 'United-States'], **slack: 4.7035**
['24' 'Private' 'Assoc-voc' 'Never-married' 'Adm-clerical' 'White' ' Male' '30' 'United-States'], **slack: 4.4904**
['61' 'Private' '9th' 'Divorced' 'Handlers-cleaners' 'White' 'Male' '40' 'Puerto-Rico'], **slack: 4.2766**
['35' 'Self-emp-not-inc' 'HS-grad' 'Never-married' 'Machine-op-inspct' 'White' 'Male' '40' 'United-States'], **slack: 3.9980**
['35' 'Private' '12th' 'Divorced' 'Craft-repair' 'White' 'Male' '50' 'United-States'], **slack: 3.9727**

Negative:
['36' 'Private' 'Doctorate' 'Married-civ-spouse' 'Prof-specialty' 'White' ' Male' ' 50' ' Canada'],

slack: 3.0918
['57' 'Private' 'Prof-school' 'Married-civ-spouse' 'Prof-specialty' 'White' 'Male' '60' 'United-States'], **slack: 2.7787**
['51' 'Self-emp-inc' 'Bachelors' 'Married-civ-spouse' 'Exec-managerial' 'Asian-Pac-Islander' 'Female' '35' 'Taiwan'], **slack: 2.7254**
['52' 'Self-emp-inc' 'Bachelors' 'Married-civ-spouse' 'Sales' 'White' 'Male' '55' 'United-States'], **slack: 2.6996**
['44' 'Private' 'Bachelors' 'Married-civ-spouse' 'Exec-managerial' 'White' 'Male' '50' 'United-States'], **slack: 2.5690**

5. C is the only hyperparameter to tune on dev set. Vary your C: 0.01, 0.1, 1, 2, 5, 10.
Report for each C: (a) training time, (b) training error, (c) dev error, and (d) number of support vectors.
Answer:

Table 1.2: Modeling performance with varying C

C	Training time (sec)	Training error (%)	Dev error (%)	Number of support vectors
0.01	3.448	17.86	16.38	2483
0.1	3.050	16.82	16.38	2083
1	3.335	15.72	16.05	1964
2	3.709	15.60	16.18	1942
5	4.773	15.44	15.85	1923
10	6.067	15.46	15.92	1917

6. Do you observe any trends from these? Are these trends consistent with the theory?
Note: I observed C = 5 resulting in the best dev error rate.
Answer: We increases the following trends and they are consistent with theory. As C increases, we are actually increasing the penalty on a wrong classification, and increasing the "complexity of the model", so the training time also increases. However, since less tolerance increases the dependency on the training data, training error decreases as C increases. However, keep increasing C is like increasing the the chance of having a "overfit" problem (the larger the C is, the more the model is dependent on the training data, but not necessarily generates well to the dev/test data), so the dev error first decreases, and then increases as C increases. It takes tuning effort to find the best C, which we found to be 5. The number of support vectors, however, decreases almost monotonically as C increases, since the larger the C is, the narrower the margin is, and the less support vectors that are margin violations/right on the margins are.

7. Make a plot from the above, with x-axis being C, and y-axis being training and dev error rates (i.e., two curves on one plot).
Answer:

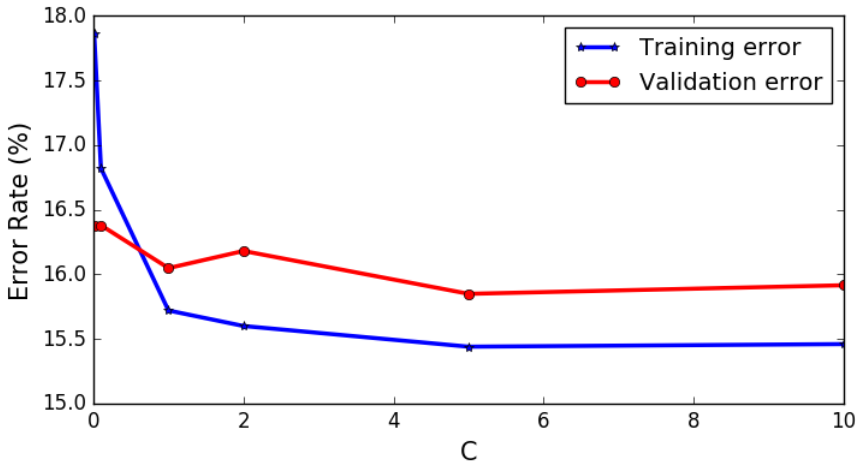

Figure 1.1: Error rate vs. varying C

8. Make a plot of training time for 5, 50, 500, and 5000 training examples (x-axis: number of examples, yaxis: seconds) (fixing C = 1). Can you figure out the time complexity of default SVM training algorithm (with respect to the number of training examples) from this figure by curve fitting?

Answer: From the plot below, we conjectures that the time complexity is $O(n^2)$. We also compare the ratios of $\frac{n}{time}$, $\frac{n^2}{time}$ and $\frac{n^3}{time}$ for $n = 5, 50, 500, 5000$, and the $\frac{n^2}{time}$ have the most similar ratios on the above sample sizes. That confirms that the time complexity is $O(n^2)$.

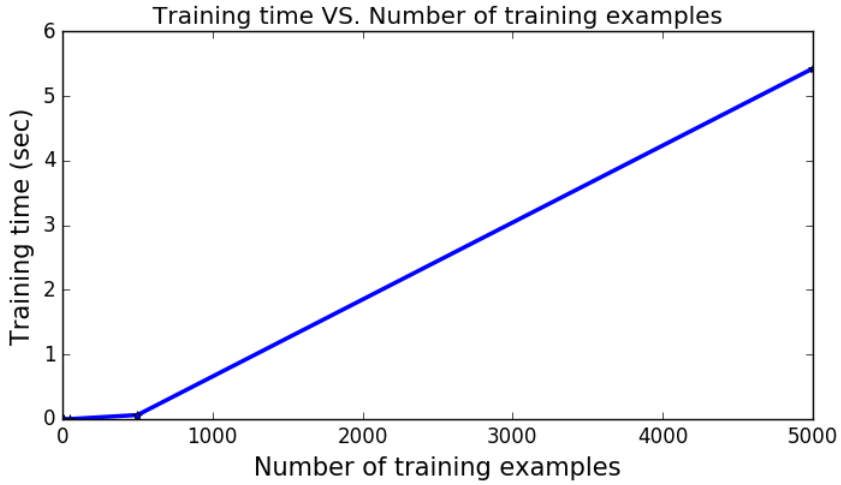

Figure 1.2: Training time vs. number of training examples

2 ONLINE SVM (PEGASOS)

1. The default SVM solver is dual and batch and you have seen how slow it is. Now implement the very simple primal and online (sub-gradient descent) Pegasos algorithm (from the slides and extra reading on the course website), for linear kernel and C = 1. The update rule is very similar to perceptron/MIRA.
Let it run until (reasonable) convergence (until the objective does not change significantly or until max. epoch). For each epoch, report (a) objective, (b) training error, (c) dev error.
Answer: The required data is in the table below. We ran Pegasos until both the objective function and dev error doesn't change much. It took 152 epochs to converge.

Epoch	Train_Err	Dev_Err	Objective	Epoch	Train_Error	Dev_Error	Objective
1	0.1910	0.2000	3990	80	0.1570	0.1630	1900
2	0.1900	0.2170	2930	81	0.1570	0.1620	1890
3	0.1780	0.1840	2340	82	0.1570	0.1620	1890
4	0.1850	0.1910	2320	83	0.1580	0.1610	1890
5	0.1690	0.1840	2140	84	0.1580	0.1610	1890
6	0.1690	0.1780	2090	85	0.1580	0.1620	1890
7	0.1690	0.1750	2040	86	0.1570	0.1620	1880
8	0.1860	0.1960	2210	87	0.1580	0.1630	1880
9	0.1870	0.1930	2220	88	0.1580	0.1640	1890
10	0.1880	0.1970	2250	89	0.1570	0.1630	1880
11	0.1800	0.1870	2120	90	0.1580	0.1620	1890
12	0.1770	0.1840	2110	91	0.1580	0.1630	1880
13	0.1950	0.2160	2340	92	0.1580	0.1620	1890
14	0.1910	0.2020	2250	93	0.1580	0.1630	1880
15	0.1870	0.1990	2230	94	0.1580	0.1620	1870
16	0.1820	0.1980	2210	95	0.1580	0.1620	1880
17	0.1900	0.2050	2300	96	0.1580	0.1620	1880
18	0.1830	0.1960	2210	97	0.1580	0.1620	1880
19	0.1800	0.1980	2180	98	0.1580	0.1620	1880
20	0.1760	0.1930	2120	99	0.1580	0.1640	1880
21	0.1780	0.1970	2150	100	0.1590	0.1620	1880
22	0.1740	0.1910	2100	101	0.1590	0.1640	1880
23	0.1690	0.1810	2040	102	0.1590	0.1640	1870
24	0.1720	0.1850	2070	103	0.1590	0.1610	1880
25	0.1660	0.1830	2030	104	0.1590	0.1630	1870
26	0.1650	0.1820	2020	105	0.1590	0.1620	1880
27	0.1700	0.1860	2060	106	0.1580	0.1630	1870
28	0.1670	0.1820	2030	107	0.1590	0.1630	1870
29	0.1650	0.1820	2000	108	0.1580	0.1620	1870
30	0.1670	0.1800	2010	109	0.1580	0.1620	1870
31	0.1610	0.1710	1960	110	0.1580	0.1620	1870

32	0.1630	0.1750	1990
33	0.1630	0.1760	1990
34	0.1590	0.1700	1960
35	0.1610	0.1710	1960
36	0.1580	0.1700	1950
37	0.1600	0.1700	1950
38	0.1620	0.1780	1970
39	0.1570	0.1660	1930
40	0.1560	0.1640	1920
41	0.1570	0.1640	1930
42	0.1570	0.1710	1950
43	0.1570	0.1680	1940
44	0.1580	0.1660	1940
45	0.1590	0.1680	1940
46	0.1580	0.1660	1930
47	0.1580	0.1670	1930
48	0.1560	0.1680	1930
49	0.1570	0.1660	1940
50	0.1570	0.1640	1930
51	0.1570	0.1640	1930
52	0.1570	0.1640	1930
53	0.1580	0.1630	1910
54	0.1570	0.1620	1920
55	0.1580	0.1650	1920
56	0.1570	0.1610	1920
57	0.1580	0.1650	1930
58	0.1580	0.1630	1920
59	0.1560	0.1620	1900
60	0.1580	0.1610	1910
61	0.1570	0.1610	1900
62	0.1570	0.1600	1900
63	0.1580	0.1620	1910
64	0.1580	0.1600	1910
65	0.1560	0.1620	1890
66	0.1580	0.1610	1910
67	0.1580	0.1610	1910
68	0.1570	0.1630	1900
69	0.1580	0.1620	1910
70	0.1580	0.1620	1900
71	0.1570	0.1640	1900
72	0.1570	0.1640	1900
73	0.1560	0.1620	1880
74	0.1560	0.1640	1900
75	0.1560	0.1630	1890
111	0.1580	0.1620	1870
112	0.1580	0.1630	1870
113	0.1580	0.1620	1870
114	0.1590	0.1620	1870
115	0.1590	0.1620	1870
116	0.1590	0.1640	1870
117	0.1590	0.1620	1870
118	0.1590	0.1620	1870
119	0.1580	0.1610	1870
120	0.1580	0.1610	1870
121	0.1590	0.1630	1870
122	0.1590	0.1610	1870
123	0.1590	0.1600	1870
124	0.1580	0.1610	1870
125	0.1580	0.1610	1870
126	0.1580	0.1610	1860
127	0.1580	0.1600	1870
128	0.1590	0.1620	1870
129	0.1570	0.1600	1860
130	0.1590	0.1610	1870
131	0.1590	0.1600	1870
132	0.1580	0.1600	1870
133	0.1580	0.1590	1870
134	0.1590	0.1600	1870
135	0.1580	0.1620	1860
136	0.1580	0.1610	1860
137	0.1580	0.1600	1870
138	0.1580	0.1620	1870
139	0.1580	0.1610	1870
140	0.1570	0.1600	1860
141	0.1570	0.1600	1860
142	0.1580	0.1610	1860
143	0.1580	0.1610	1870
144	0.1580	0.1600	1860
145	0.1570	0.1590	1860
146	0.1580	0.1610	1860
147	0.1570	0.1600	1860
148	0.1570	0.1600	1860
149	0.1570	0.1600	1860
150	0.1580	0.1600	1860
151	0.1580	0.1600	1860
152	0.1570	0.1600	1860

76	0.1570	0.1620	1900
77	0.1570	0.1610	1900
78	0.1570	0.1620	1890
79	0.1570	0.1620	1890

Table 2.1: Error Rate And Objective Function Changes During Pegasos

2. Does the objective eventually converge to the one returned by sklearn? Is yours faster or slower?

Answer: Not exactly, but close. The objective function in our Pegasos implementation converges at 1861.5, whereas our objective function with linear kernel is 1840.2. In theory, those two should converge to a very similar number, and we think our results to be similar enough.

Our implementation of Pegasos is significantly slower than SVM from sklearn, though. SVM takes about 3 seconds to run, whereas our Pegasos takes about 28 seconds. We know that the default SVM is dual and batch, whereas Pegasos is primal and online. It could be that SVM from sklearn implements special optimization schemes, or we could improve our implementation to be faster. In general people seem to expect Pegasos to be faster than SVM, but it could be the case that Pegasos would scales better on a larger dataset, since we assume its time complexity could be $O(n)$, though not entirely sure.

3. Make two plots from the above experiment: the objective vs. epoch, and train/dev error rates vs. epoch.

Answer:

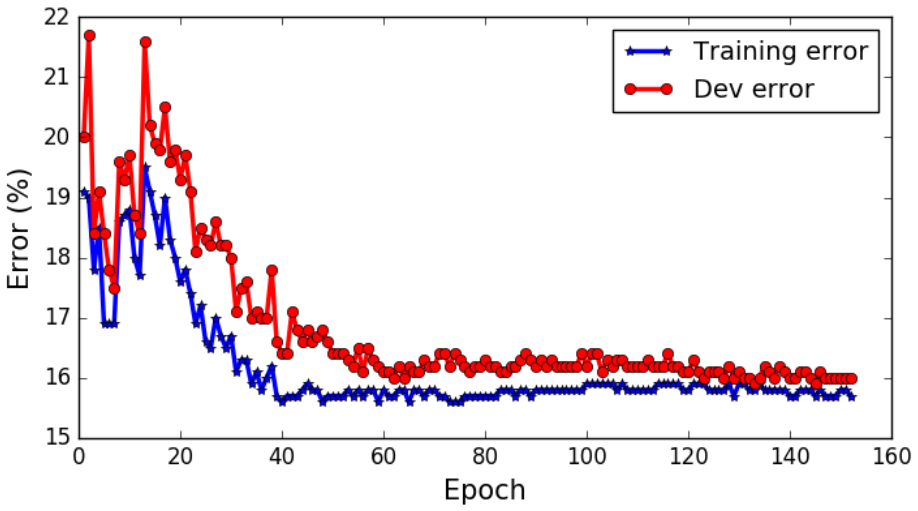

Figure 2.1: Online Pegasos algorithm performance

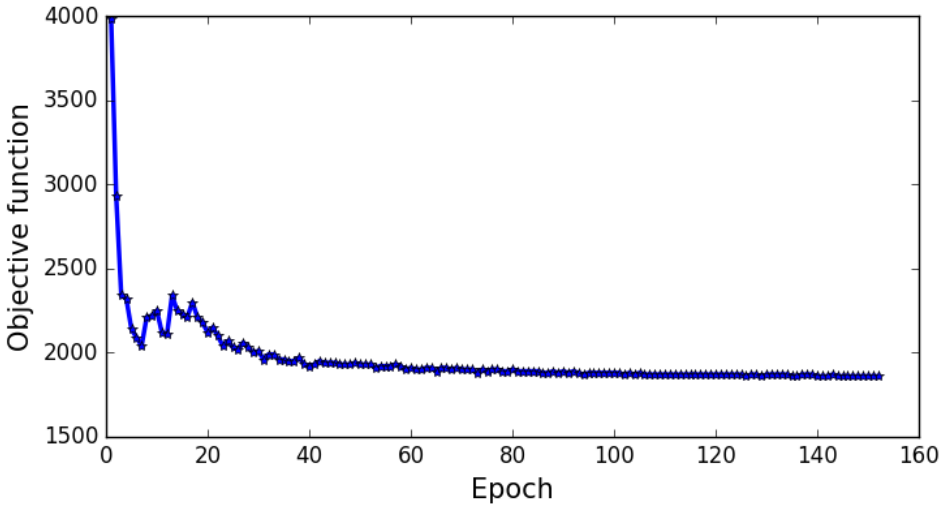

Figure 2.2: Online Pegasos algorithm objective function

We can see that both training/dev errors and objective functions converges after about 150 epochs.

4. How many support vectors are there in the final model? How does it compare with sklearn.

Answer: 2504. This is much more than what sklearn has 1005 + 959 = 1964. And that makes sense: Pegasos has an online updating scheme whereas SVM has an "batch" one. So vectors that involves an "special update (when margin violation on the current example happens)" in Pegasos are much more than the support vectors in SVM, which has a batch updating scheme.

3 Quadratic Kernels

1. In sklearn, replace linear kernel with quadratic kernel (kernel='poly', degree=2, coef0=1). Train with C = 1. Report the training time, and train/dev error rates.

Answer: Training time is 3.183 seconds when C = 1. Training error is %18.28, and dev error rate is %16.45.

2. Why do we need to set coef0=1?

Answer: To make sure that there is a "bias" feature in the quadratic kernel. This is not an issue if you use the linear kernel, but is worth attention when using a polynomial kernel with degree more than 1. It's just how sklearn.SVM is written.

3. Tune C in a way you see fit. Report for each C: (a) training time, (b) training error, (c) dev error, and (d) number of support vectors. Note: I observed (significantly) better train/dev error rates than linear kernel by tuning C.

Answer:

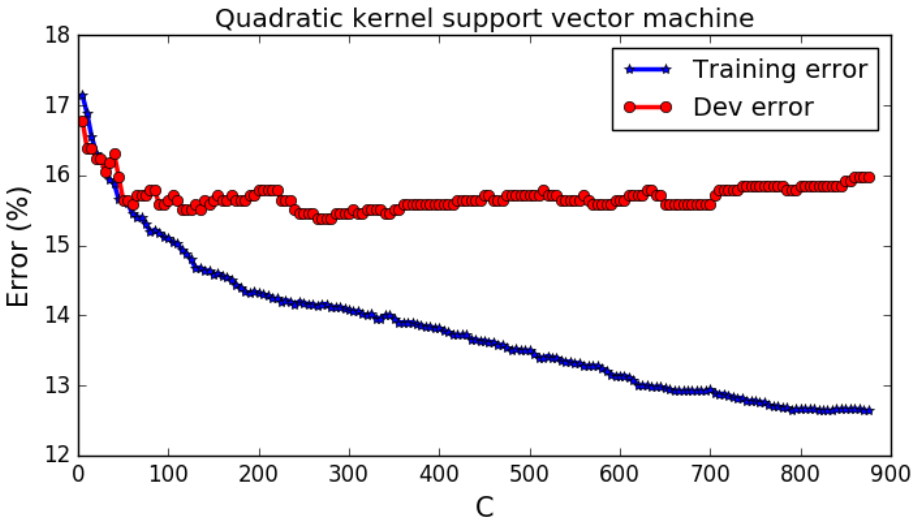

Figure 3.1: C tuning

We wrote a program to tune C, and recorded the data along the way. From the figure 3.1 we can conclude that when $C = 265 \leq c \leq 280$, the dev error can get as low as 15.38% at quadratic kernel, whereas the best dev error is 15.85% at linear kernel. The other "good range" we observe is $C = 115 \leq c \leq 128$, and the dev error is 15.52 %. However, the predicted positive rate is comparable on both range, so we decided to use $C = 270$ as our final answer.

C	TrainTime(sec)	TrainErr	DevErr	#SVC	C	TrainTime(sec)	TrainErr	DevErr	#SVC
5	5.306	17.14	16.78	2174	455	6.844	13.62	15.72	1982
10	5.079	16.88	16.38	2097	460	6.912	13.62	15.65	1982
15	5.081	16.56	16.38	2058	465	7.171	13.58	15.65	1981
20	5.011	16.32	16.25	2036	470	6.844	13.58	15.65	1980
25	5.043	16.24	16.25	2027	475	6.870	13.54	15.72	1979
30	5.065	16.10	16.05	2031	480	7.975	13.5	15.72	1981
35	5.044	15.94	16.18	2018	485	8.148	13.52	15.72	1982
40	5.063	15.88	16.31	2011	490	8.417	13.5	15.72	1982
45	4.976	15.66	15.98	2006	495	7.896	13.5	15.72	1982
50	4.980	15.64	15.65	1997	500	7.743	13.5	15.72	1978
55	5.042	15.60	15.65	1993	505	7.915	13.44	15.72	1978
60	5.110	15.46	15.58	1987	510	7.963	13.4	15.72	1979
65	5.149	15.40	15.72	1988	515	7.941	13.4	15.78	1981
70	5.169	15.40	15.72	1991	520	7.089	13.42	15.72	1980

75	5.155	15.30	15.72	1986	525	7.348	13.4	15.72	1980
80	5.290	15.20	15.78	1984	530	7.812	13.4	15.72	1977
85	5.228	15.22	15.78	1979	535	7.133	13.36	15.65	1977
90	5.296	15.18	15.58	1974	540	7.162	13.34	15.65	1979
95	5.393	15.12	15.58	1974	545	7.380	13.34	15.65	1980
100	5.395	15.10	15.65	1976	550	7.124	13.32	15.65	1984
105	5.337	15.04	15.72	1975	555	7.445	13.32	15.65	1986
110	5.367	15.02	15.65	1972	560	7.105	13.28	15.72	1989
115	5.477	14.94	15.52	1970	565	7.171	13.28	15.65	1986
120	5.450	14.88	15.52	1966	570	7.181	13.28	15.58	1987
125	5.510	14.80	15.52	1967	575	7.206	13.28	15.58	1986
130	5.488	14.68	15.58	1968	580	7.248	13.24	15.58	1988
135	5.528	14.68	15.52	1971	585	7.186	13.2	15.58	1988
140	5.497	14.64	15.65	1972	590	7.310	13.16	15.58	1988
145	5.574	14.64	15.58	1965	595	7.288	13.14	15.65	1988
150	5.663	14.58	15.65	1968	600	8.273	13.14	15.65	1988
155	5.771	14.60	15.72	1961	605	7.501	13.14	15.65	1988
160	5.584	14.56	15.65	1959	610	8.466	13.12	15.72	1990
165	5.658	14.54	15.65	1964	615	9.114	13.08	15.72	1989
170	5.694	14.50	15.72	1964	620	8.350	13	15.72	1990
175	5.790	14.44	15.65	1966	625	7.605	13	15.72	1990
180	5.748	14.40	15.65	1968	630	8.031	13	15.78	1988
185	6.219	14.34	15.65	1971	635	7.961	12.98	15.78	1990
190	6.347	14.32	15.72	1969	640	7.427	12.98	15.72	1988
195	5.827	14.34	15.72	1969	645	7.576	12.98	15.72	1991
200	5.846	14.32	15.78	1970	650	7.720	12.96	15.58	1990
205	5.762	14.30	15.78	1972	655	7.548	12.94	15.58	1990
210	5.871	14.28	15.78	1971	660	7.435	12.92	15.58	1989
215	5.893	14.24	15.78	1971	665	7.586	12.92	15.58	1989
220	5.995	14.24	15.78	1969	670	7.754	12.92	15.58	1989
225	5.961	14.20	15.65	1968	675	7.572	12.92	15.58	1992
230	5.955	14.22	15.65	1966	680	7.823	12.92	15.58	1994
235	6.137	14.20	15.65	1966	685	7.607	12.92	15.58	1993
240	5.996	14.16	15.52	1968	690	7.774	12.92	15.58	1993
245	6.016	14.20	15.45	1969	695	7.733	12.92	15.58	1995
250	6.120	14.18	15.45	1967	700	7.660	12.94	15.58	1996
255	6.052	14.16	15.45	1967	705	7.771	12.9	15.72	1998
260	6.075	14.16	15.45	1964	710	7.842	12.88	15.78	1995
265	6.519	14.14	15.38	1965	715	7.767	12.88	15.78	1996
270	6.399	14.16	15.38	1966	720	8.440	12.86	15.78	1996
275	6.353	14.16	15.38	1962	725	8.114	12.84	15.78	1994
280	6.446	14.12	15.38	1966	730	8.150	12.82	15.78	1994
285	6.186	14.12	15.45	1964	735	8.071	12.82	15.85	1994
290	6.251	14.12	15.45	1969	740	9.226	12.78	15.85	1994

295	6.276	14.10	15.45	1967	745	8.609	12.78	15.85	1990
300	6.387	14.08	15.45	1967	750	7.787	12.78	15.85	1991
305	6.502	14.06	15.52	1970	755	7.662	12.76	15.85	1992
310	6.585	14.06	15.45	1969	760	7.694	12.76	15.85	1991
315	6.406	14.02	15.45	1969	765	7.792	12.72	15.85	1992
320	6.696	14.00	15.52	1967	770	5.755	12.7	15.85	1991
325	7.239	14.02	15.52	1968	775	4.839	12.7	15.85	1990
330	6.769	13.96	15.52	1965	780	4.861	12.68	15.85	1989
335	6.812	13.96	15.52	1966	785	4.821	12.68	15.78	1991
340	6.607	14.00	15.45	1965	790	4.795	12.64	15.78	1991
345	6.710	14.00	15.45	1967	795	4.832	12.66	15.78	1991
350	6.743	13.96	15.52	1967	800	4.829	12.66	15.85	1995
355	6.663	13.90	15.52	1972	805	4.864	12.66	15.85	1993
360	6.499	13.90	15.58	1973	810	4.813	12.66	15.85	1994
365	6.686	13.90	15.58	1975	815	4.893	12.66	15.85	1993
370	6.694	13.90	15.58	1974	820	4.884	12.64	15.85	1995
375	6.788	13.88	15.58	1978	825	4.944	12.64	15.85	1997
380	6.811	13.86	15.58	1977	830	5.653	12.64	15.85	1997
385	6.617	13.84	15.58	1978	835	4.972	12.64	15.85	1996
390	7.097	13.84	15.58	1975	840	4.911	12.66	15.85	1997
395	6.579	13.82	15.58	1980	845	4.911	12.66	15.85	1995
400	6.861	13.82	15.58	1978	850	4.925	12.66	15.92	1995
405	7.481	13.78	15.58	1980	855	4.899	12.66	15.92	1993
410	7.380	13.76	15.58	1980	860	4.905	12.66	15.98	1993
415	6.862	13.72	15.58	1978	865	4.950	12.66	15.98	1992
420	8.124	13.72	15.65	1977	870	4.954	12.64	15.98	1989
425	7.245	13.72	15.65	1980	875	4.906	12.64	15.98	1985
430	7.429	13.72	15.65	1978					
435	6.978	13.66	15.65	1981					
440	6.875	13.66	15.65	1981					
445	6.848	13.64	15.65	1981					
450	7.194	13.64	15.72	1983					

Table 3.1: C tuning process

4. Do you observe any trends from these? Are these trends consistent with the theory?
Answer: Yes. The training error keeps decreasing as C increases, whereas the dev error first decreases, and then stays stable for a while, and starts to increase after that. This is because small C is allowing more error, whereas big C is restricting the decision service more strictly according to the data. So if C is too small, it's kind of like "underfitting", and C too big is like "overfitting". It takes tuning effort to find the "sweet spot" in between.

5. Why you can't access clf.coef_ any more?
Answer: Because we are computing dot product between $\boldsymbol{x}$'s, and then square them, instead

of first square every x_i vector, and then square them. So we are not actually computing $\phi(\boldsymbol{x_i})$ explicitly since that would be too slow, and hence cannot access the coefficients of the primal any more.

6. Can you adapt Pegasos for quadratic kernels? (Just describe the algorithm, but no need to implement it!)
Answer: Instead of working on the primal problem, we are working on the dual problem in Pegasos for quadratic kernels, to take advantage of the speed-up using kernels instead of explicitly compute $\phi(\boldsymbol{x_i})$'s. That is, we are updating α_i's instead of updating w. Initially, we set the $\boldsymbol{\alpha}$ vector to be 0. Then pick the i th example from the dataset at random for each t th iteration, and update the α_i that is corresponding to that example only, leaving all the other α_j, $j \neq i$ to be the same as in last iteration. Similar as the updating rule in linear kernel Pegasos, we have a variable learning rate $\frac{1}{\lambda t}$, where $\lambda = \frac{2}{NC}$. If $y_i \frac{1}{\lambda t} \sum_j \alpha_j y_i K(\boldsymbol{x_i}, \boldsymbol{x_j}) < 1$, $\alpha_i = \alpha_i + 1$. Else we don't update α_i and move to the next iteration. Here $K(\boldsymbol{x_i}, \boldsymbol{x_j}) = (\boldsymbol{x_i} \cdot \boldsymbol{x_j})^2$, and the computation complexity reduce to $O(d)$ instead of $O(d^2)$, as in computing $\phi(\boldsymbol{x})$ (squaring the features) first and then dot product.

4 FINAL

Collect your best model and predict on income.test.txt to income.test.predicted. The latter should be similar to the former except that the target (>=50K, <50K) field is added. Q: what's your best error rate on dev, and which algorithm (and settings) achieved it?

Answer: Our best dev error rate is %15.38, and SVM with quadratic kernel when C = 270 achieved it.

Q: what's your % of positive examples on dev and test according to this best model?
Answer: %20.62 on dev set and %21.07 on test set.