BUSN 20800: Big Data

Lecture 4: Classification

Dacheng Xiu

University of Chicago Booth School of Business

Classification

- K-nearest neighbors and group membership.
- Binary classification: from probabilities to decisions.
- misclassification, sensitivity and specificity.
- Multinomial logistic regression: fit and probabilities.

Classification

Just as in linear regression, we have a set of training observations $(x_1, y_1), ..., (x_n, y_n)$. But now y_i are qualitative rather than quantitative, i.e. y is membership in a category $\{1, 2, ..., m\}$.

The classification problem: given new x_i^{new} what is the class label $f(x_i^{new})$?

The quality of classifier can be assessed by its missclassification risk, i.e. probability of falsely classifying a new observation

$$P(Y^{new} \neq f(x_i^{new}))$$

This quantity is unknown but can be estimated by a proportion of wrong labels in a validation dataset. Good classifiers yield small risk.

Bayes Classifier

There is actually a theoretically optimal classifier, the *Bayes classifier*, which minimizes the misclassification risk.

The idea is to assign each observation to *the most likely class, given its predictor values*, i.e. choose the class $j \in \{1, 2, ..., m\}$ for which

$$P(Y = j|X = x)$$

is the largest.

Unfortunately $P(Y = j|X = x)$ is not known. Bayes classifier is unattainable gold standard. But! We can estimate it!

Analogy to regression

	Classification	Regression
Risk	classification risk $P(Y \neq f(X))$	squared error risk $E(Y - f(X))^2$
Minimizer	Bayes classifier $f(x) = \arg\max_j P(Y = j \mid X = x)$	conditional expectation $f(x) = E(Y \mid X = x)$

Classifiers

There are many ways to estimate $P(Y = j|X = x)$ from the training data.

- We can go *parametric*:
  - Assume that $P(Y = j|X = x, \beta)$ is a specific function of unknown parameters β and learn those.
  - Sounds familiar? Logistic regression
- We can go *non-parametric*:
  - We estimate $P(Y = j|X = x)$ directly without estimating any parameters.
  - K-nearest Neighbors (KNN)

Nearest Neighbors

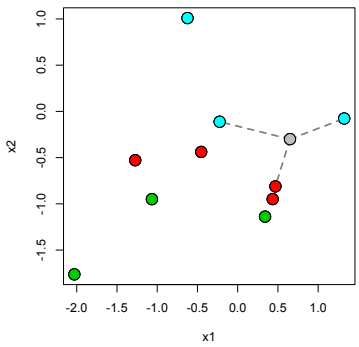
The idea is to estimate $P(Y = j|X = x_{new})$ locally by looking at the labels of similar observations that we already saw.

KNN: what is the most common class around x?

1. Take K-nearest neighbors $\mathbf{x}_{i_1} \dots \mathbf{x}_{i_k}$ of x_{new} in the training data
   Nearness is in euclidean distance: $\sqrt{\sum_{j=1}^{p}(x_j - x_{ij})^2}$.
2. Estimate
$$\widehat{P}(Y = j|X = x_{new}) = \frac{1}{K}\sum_{k=1}^{K}\mathbf{1}_{\{y_{i_k}=j\}}.$$
3. Select the class with highest $\widehat{P}(Y = j|X = x_{new})$ (Bayes classifier).

Since we're calculating distances on X, scale Matters!
We'll use Python's StandardScaler function in sklearn to divide each x_j by $\mathrm{sd}(x_j)$
The new units of distance are in *standard deviations*.

Nearest Neighbors



K-NN's collaborative estimation:

Each neighbor votes.

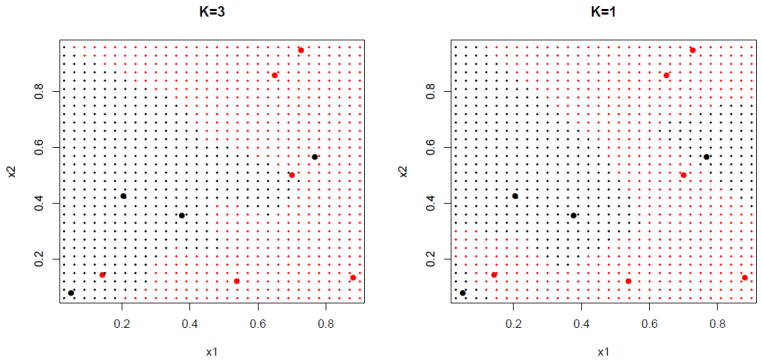
Neighborhood is by shortest distance (shown as the dashed lines).

The relative vote counts provide a very crude estimate of probability.

For 3-nn, $\mathrm{p}(\text{blue}) = 2/3$, but for 4-nn or 2-nn, it's only $1/2$.

Sensitive to neighborhood size (think about extremes: *1* or n).

Nearest Neighbors: Decision Boundaries



- Larger K leads to higher training error (proportion of in-sample misclassification rate)
- Smaller K leads to higher flexibility (overfitting and poor out-of-sample misclassification rate)

Nearest neighbors in Python (implementation)

Compute the euclidean distance of X_{new} in the training data

```
def euc_distance(arr1, arr2):
    distance = np.sqrt(sum((arr1-arr2)**2))
    return distance
```

Sort by distance, find the index of the closest K neighbors and select the class with highest possibility.

```
def knn_classifier(X, y, testVector, k):
    distance_list = [euc_distance(testVector, x) for x in X]
    neighbors = np.argsort(distance_list)[:k]
    count = Counter(y[neighbors][0])
    return count.most_common()[0][0]
```

Nearest neighbors in Python (package)

Alternatively, you may use some existing package, which is less transparent but offers more features. The `sklearn` package includes function `KNeighborsClassifier`.

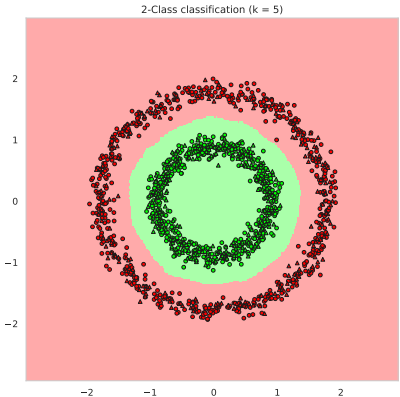
You set `n_neighbors` to specify how many neighbors get to vote.

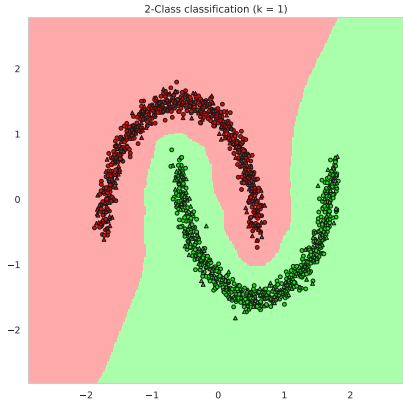
```
from sklearn.neighbors import KNeighborsClassifier

nn1 = KNeighborsClassifier(n_neighbors=1, n_jobs=-1).fit(X_train, y_train)
nn5 = KNeighborsClassifier(n_neighbors=5, n_jobs=-1).fit(X_train, y_train)
y_pred_1 = nn1.predict(X_test)
y_pred_5 = nn5.predict(X_test)
```

You set `n_jobs` to use parallel computing.

KNN with simulated data





Lending Club Data (Revisit HW 2)

LendingClub (LC) is a US peer-to-peer lending company. It was the first peer-to-peer (P2P) lender to register its offerings as securities with the Securities and Exchange Commission (SEC), and to offer loan trading on a secondary market.

In this case study, we understand how data is used to minimize the risk of losing money while lending to customers in P2P business.

We consider 2 possible loan status

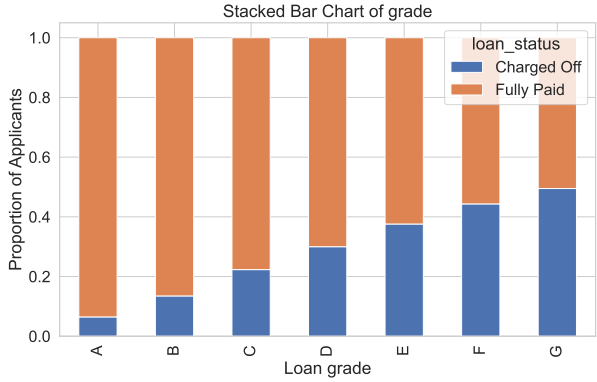
`Fully paid(0)`: Applicant has fully paid the loan

`Charged-off(1)`: Applicant has defaulted on the loan

We have 27 explanatory variables, 15 of which are categorical variables.

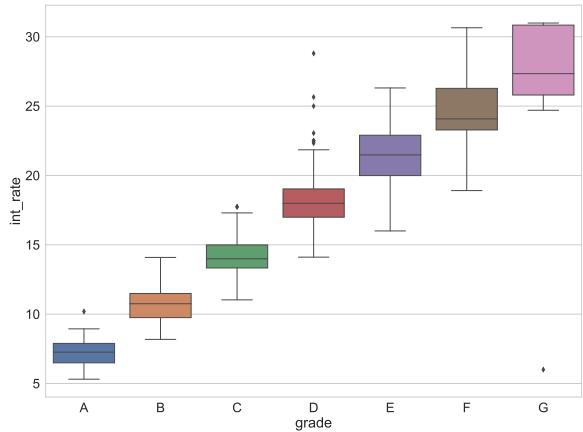
We use 07-18 data as training sample and 19-20 data as test sample.

Some variables are special: grade



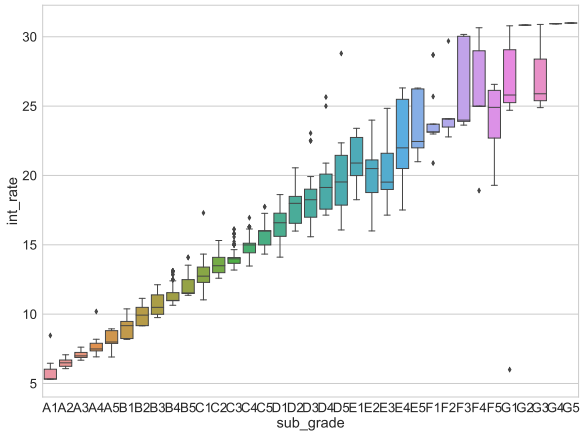
As grade varies from A to G, the default probability increases.

Some variables are special: interest rate vs grade



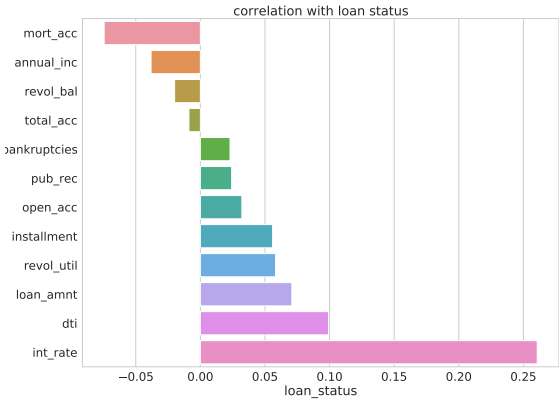
As default probability increases, interest rate increases. Both are determined by LC.

Some variables are special: interest rate vs subgrade



The same is true for subgrade. We will use these variables to compare with LC's model.

Correlations with default

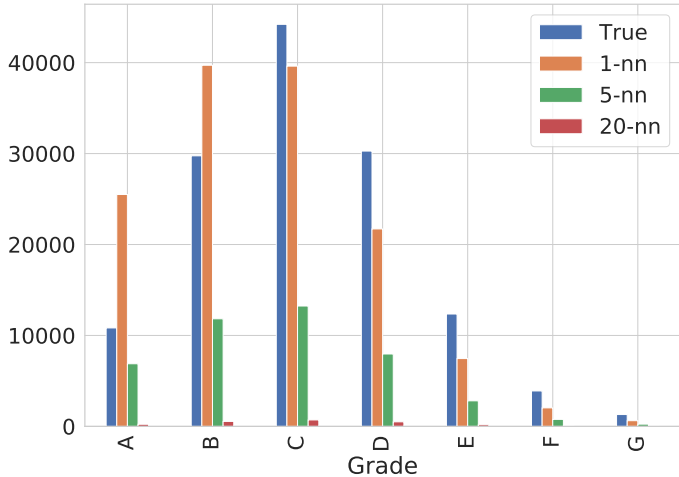


Some covariates are better discriminators (dti, mort_acc).

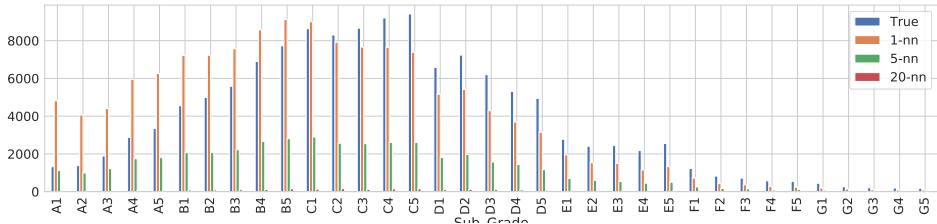
mort_acc: number of mortgage accounts

dti: borrower's debt-to-income ratio

Predicted # of Defaults by Grade in Testing Sample



Predicted # of Defaults by Subgrade in Testing Sample



5-nn and 20-nn clearly underestimate # of default, but in terms of accuracy, they dominate 1-nn!

Accuracy ($P(\widehat{Y} = Y)$): 1-nn: 0.69, 5-nn: 0.77, 20-nn: 0.81

KNN: Pros and Cons

Pros

- KNN's are simple
- KNN's naturally handle multiple categories ($m > 2$)
- KNN's will outperform linear classifiers when the decision boundary is non-linear

Cons

- Computing neighbors can be costly for large n and p.
- KNN's do not perform variable selection. How did we avoid the curse of dimensionality?
- Choosing K can be tricky. Cross-validation works, but is unstable.
- And the classification is very sensitive to K.
- All you get is a classification, with only rough local probabilities. Without good probabilities we cannot assess uncertainty.
- There is no natural or simple way to handle categorical variables.

Binary classification via logistic regression

Many decisions can be reduced to binary classification. $y_i \in \{0, 1\}$

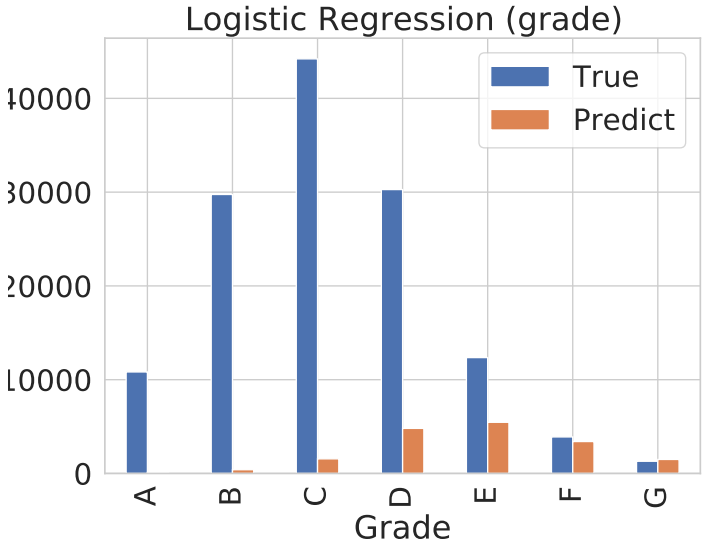
KNN's were an example of a *non-parametric* classification method.

A useful parametric alternative for two categories is the *logistic regression*.

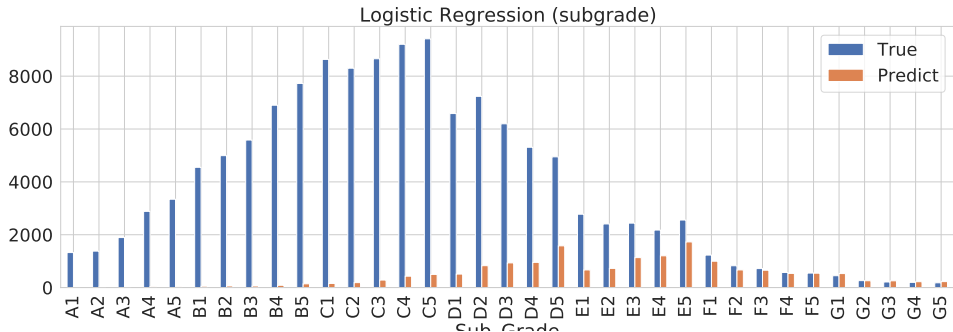
Compared to KNN's

- *Logistic regression* yields parametric decision boundaries (linear, quadratic depending on our regression equation) $\implies$ it is principled but it can be flexible
- *Logistic regression* is a 'global' method, i.e. it uses all the training data to estimate probabilities, not just neighbors $\implies$ probability estimates are more stable
- *Logistic regression* can do variable selection! (yay!)

Logistic regression



Logistic regression



Accuracy ($P(\widehat{Y} \neq Y)$): Logit: 0.81

Decision making

Logistic regression gives us an estimate of $P(Y = 1|X = x, \beta)$.

The Bayes decision rule is based purely on probabilities: classify as a defaulter, i.e., $\widehat{Y} = 1$, when

$$P(Y = 1|X = x, \widehat{\beta}) > 0.5.$$

Recall that the Bayes rule is optimal in terms of misclassification error $P(\widehat{Y} \neq Y)$.

Decision making

There are two ways to be wrong in a binary problem.

$$P(\widehat{Y} \neq Y) = P(\widehat{Y} = 1|Y = 0)P(Y = 0) + P(\widehat{Y} = 0|Y = 1)P(Y = 1).$$

- False positive (Type I error): predict $\widehat{Y} = 1$ when $Y = 0$.
  (classify as defaulters when they are not)
- False negative (Type II error): predict $\widehat{Y} = 0$ when $Y = 1$.
  (classify as non-defaulters when they in fact are)

Misclassification error weights different errors by the relative weights of different labels in the population.

Asymmetric error importance and imbalanced class proportion

Both false positives and false negatives are bad, but sometimes one of them can be much worse $\implies$ the cost *can be asymmetric*!

- investment
- medical diagnosis

Maximizing accuracy does not appear to be the most relevant criterion in practice.

In many scenarios in practice, the class label for worse outcome is also rare!

- fraud detection
- default prediction
- rare disease

Maximizing accuracy puts more weight on the larger class. $P(Y = 0)/P(Y = 1)$ is large $\implies$ far more false negatives!

Minimizing cost

Say that, on average, for every 1$ loaned lenders make 25¢ in interest if it is repayed but lose 1$ if they default.

This gives the following action-*cost* matrix

	payer	*defaulter*
loan	-0.25	1
no loan	0	0

Suppose you estimate p for the probability of default.
Expected *profit* from lending is greater than zero if

$$(1-p)\frac{1}{4} - p > 0 \quad \Leftrightarrow \quad \frac{1}{4} > \frac{5}{4}p \quad \Leftrightarrow \quad p < 1/5$$

From this simple matrix you should lend whenever probability of default is less than 0.2.

Confusion matrix

An alternative approach is to think about False/True Positive Rates in tandem when evaluating a given classification rule.

We can use a confusion matrix to show information about actual and predicted classifications done by a classifier.

		Predicted	
		Negative (**N**) -	Positive (**P**) +
Actual	Negative -	True Negatives (**TN**)	False Positives (**FP**) **Type I error**
	Positive +	False Negatives (**FN**) **Type II error**	True Positives (**TP**)

Confusion matrix in Python

Load the `sklearn` package which includes function `confusion_matrix`.

`y_test` and `y_pred` are true values and predicted values.

```
from sklearn.metrics import confusion_matrix
confusion_matrix = confusion_matrix(y_test, y_pred)
```

```
5-nn
array([[520158,  33922],
       [122799,   9814]])

20-nn
     array([[552415,  1665],
            [ 132052,  561]])

logit
array([[544898,   9182],
       [124564,   8049]])
```

FP and FN rates

False Positive Rate: # misclassified as pos / # actual neg. ($\frac{FP}{FP+TN}$)

False Negative Rate: # misclassified as neg / # actual pos.($\frac{FN}{FN+TP}$)

```
# False Positive Rate
FPR = FP/(FP+TN)  5-nn: 0.06    20-nn: 0.003    logit: 0.017
# False Negative Rate
FNR = FN/(FN+TP)  5-nn: 0.926    20-nn: 0.995    logit: 0.939
```

We can choose a threshold to control false positive rate, while minimizing false negative rate.

Sensitivity and Specificity

Two more common classification rates are

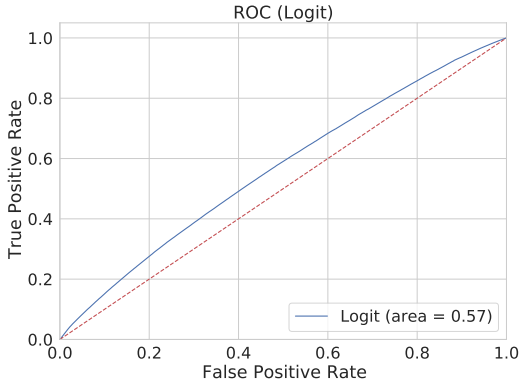
sensitivity: proportion of true $y = 1$ classified as such. (aka True Positive Rate)

specificity: proportion of true $y = 0$ classified as such. (aka True Negative Rate)

A rule is sensitive if it predicts 1 for most $y = 1$ observations, and specific if it predicts 0 for most $y = 0$ observations.

All our rules are similarly specific, though not sensitive at all. This has to do with the fact that the majority class (nearly 80%) us non-defaulters.

The ROC curve: sensitivity vs 1-specificity



Receiver Operating Characteristic: performance measure of a classifier as cutoff varies
A tight fit has the curve forced into the top-left corner.

AUC

The Area Under the Curve (AUC) is the measure of the ability of a classifier to distinguish between classes and is used as a summary of the ROC curve.

- AUC $= 1$: perfect separation.
- The higher the AUC, the better the classifier.
- $0.5 <$ AUC < 1: acceptable.
- AUC $= 0.5$: either random guessing or constant for all.
- AUC $= 0$: all predictions are off.

Multinomial Logistic Regression

Probabilities are the basis for good cost-benefit classification.

Similarly as in logistic regression (m=2), can get class probabilities $P(Y = j|X = x)$ for more than two categories($m > 2$)?

Yes! $\rightarrow$ *Multinomial Logistic Regression*

We need m models (for each category)

$$\begin{aligned} P(Y = 1|X = x) &\propto f(x^{'}\beta_1) \\ P(Y = 2|X = x) &\propto f(x^{'}\beta_2) \\ &... \\ P(Y = m|X = x) &\propto f(x^{'}\beta_m) \end{aligned} \tag{1}$$

We need to find regression coefficients β_k for each class.

We need to make sure that $\sum_{j=1}^{m} P(Y = j|X = x) = 1$

Multinomial Logistic Regression

Extend logistic regression via the multinomial logit:

$$\mathrm{p}(Y_i = j | X_i = x) = \mathrm{p}(y_j = 1 | X_i = x) = p_{ij} = \frac{e^{x_i{}' \beta_j}}{\sum_{k=1}^{K} e^{x_i{}' \beta_k}}$$

Note separate coefficients for each class: β_k.

Denote by k_i the class of i^{th} observation Y_i . Then, the likelihood is

$$LHD(\beta_1, ..., \beta_m) \propto \prod p_{ik_i}$$

and the deviance is

$$Dev(\beta_1, ..., \beta_m) \propto -2 \sum_i \log(p_{ik_i})$$

Multinomial Logistic Regression

Once we have a model, we can do variable selection in each of the m regressions.

We can use the LASSO penalty: penalized deviance minimization.

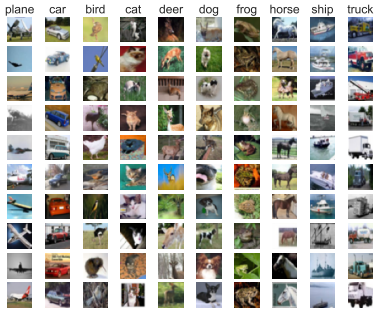
$$\min \left\{ -\frac{2}{n} \sum_i \log p_{ik_i}(\boldsymbol{\beta}) + \sum_k \sum_j \lambda |\beta_{kj}| \right\}$$

We can also have λ_k: different penalty for each class.

We can find out which predictors in x_i are relevant discriminators of each of the m classes.

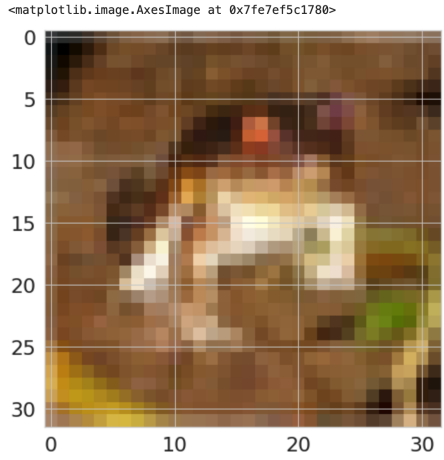
CIFAR10 Data (Revisit Lecture 1)

CIFAR-10 dataset consists of 60,000 tiny images that are 32 pixels high and wide. Each image is labeled with one of 10 classes. These 60,000 images are partitioned into a training set of 50,000 images and a test set of 10,000 images.



Suppose now that we are given a subset of CIFAR-10 training set of 5,000 images, and we wish to label a subset of 1,000 test samples.

Vectorizing images



A colored image is represented by a 32×32 matrix with 1024 pixels. Each pixel has 3 channels representing R, G, B.

We thereby can represent each image by a 3,072 dimensional vector.

Multinomial Logistic Regression

Fit the model in `LogisticRegression` function with *multi_class* $='$ *multinomial*$'$.

Assess accuracy using classification_report

```
MLR = LogisticRegression(multi_class='multinomial', solver='saga',random_state=0)
MLR.fit(X_train, y_train)
y_pred_MLR = MLR.predict(X_test)
```

We can use LASSO penalty by setting *penalty* $='$ *l*1$'$

```
MLR_LASSO = LogisticRegression(multi_class='multinomial', penalty='l1', solver='saga')
MLR_LASSO.fit(X_train, y_train)
y_pred_MLR_LASSO = MLR_LASSO.predict(X_test)
```

Comparison of prediction and truth based on testing images

Accuracy: top 1: 0.334 top five: 0.846

Interpreting the MN logit

We're estimating a function that sums to one across classes.
But now there are K categories, instead of just two.

The log-odds interpretation now compares between classes:

$$\log\left(\frac{p_a}{p_b}\right) = \log\left(\frac{e^{x'\beta_a}}{e^{x'\beta_b}}\right) = x'[\beta_a - \beta_b].$$